Formal modelling
of reactive hardware/software systems

J.P.M. Voeten, P.H.A. van der Putten, M.C.W. Geilen, M.P.J. Stevens

Abstract—The inherent complexity of modern reactive hardware/software systems requires the creation of system models before these systems are actually being built. A system model must allow a designer to reason about the essential properties of the system at the right level of abstraction. Therefore these models must be expressed in a formal language that is able to express these properties and that allows them to be examined by automated tools. In this paper we introduce a new system-level specification language called POOSL (Parallel Object-Oriented Specification Language). The key feature of the language is the expressive power to model very complex dynamic (communication and functional) behaviour as well as static (architecture and topology) structure in an object-oriented fashion. The language integrates a process part, based on the process algebra CCS, with a data part, based on the concepts of traditional object-oriented programming languages. Unlike many modelling languages today, POOSL is equipped with a complete formal mathematical semantics. Currently a number of automated software tools (model editing, simulator and compiler tools) are available.

Keywords—formal specification, object-oriented methods, hardware/software design, real-time systems.

I. INTRODUCTION

Industry struggles with mastering the development process of complex information processing systems such as data/telecommunication systems, signal and image processing systems, multimedia systems, medical systems and industrial control systems. Many of these systems can be characterised as reactive hardware/software systems. Reactive systems are systems that persistently communicate with their environment [1]. They perform very complex behaviour and are often based on communicating processes that operate in parallel. Communications take various forms such as synchronous, asynchronous, continuous and interrupt-driven. Interrupts are high-priority messages that interfere or abort the current course of behaviour. Modern reactive systems further perform complex operations on intricate data structures. Operations and communications have to satisfy stringent timing requirements. Reactive systems are often considered to be software running on a single embedded processor. Reactive hardware/software systems, on the other hand, are implemented by a mix of hardware and software components. In addition to the constraints of general reactive systems, they have to satisfy stringent architectural constraints that impose physical distribution, physical topology and hardware/software partitioning.

Currently, industry is facing a crisis in the design of complex systems [2], [3]. There are a number of reasons for this. Due to their combination of properties reactive hardware/software systems are just inherently complex to design, verify, validate, test and debug. Further, today’s systems integrate more and more functionality and this functionality has to satisfy stringent requirements of quality, flexibility, reliability and reusability. In addition, deciding on appropriate implementation technology has become increasingly difficult due to the diversity of available technology components (micro processors, ASICs, ASIPs, DSPs, FPGAs, etcetera). Last but not least, industry is struggling with fast innovation speed and tight bounds on time-to-market and time-to-quality.

Existing specification and design approaches are no longer keeping pace with the growth of systems complexity. Using traditional design approaches, industry is not able to simulate, validate or verify the current generation of systems [2]. To cope with complexity, industry is forced to look at alternative higher-level approaches towards specification and design [2]. Currently the EDA industry is trying to define a new standard language, called System Level Design Language (SLDL), that is to be used at higher levels of abstraction, before hardware/software partitioning and technology choices have been made.

In this paper we introduce a new system-level modelling, specification and design language POOSL (Par-
parallel Object-Oriented Specification Language). In Section II our requirements for a system-level language are pointed out. Section III explains the basic concepts of POOSL. In Section IV we show how the language can be used to model a non-trivial part of a protocol. Section V describes the software tools that are currently available. Finally in Section VI we present our conclusions and directions for future research.

II. SYSTEM LEVEL LANGUAGE REQUIREMENTS

Figure 1 shows the role of system-level languages and methods in the design trajectory for reactive hardware/software systems [3]. It visualises the different design phases and indicates for each phase the relevant keywords and key activities. It should be clear that system-level languages (and methods) are especially useful in the early phases of design, where a system is being analysed and specified and where the system-level architecture is being decided upon. They should help to bridge the gap between requirements capture and synthesis. Therefore they should be intuitive, applicable within multidisciplinary design and easy to understand. For this reason, a system-level language should have an imperative nature [4]. During the architecture exploration phase, the implementation technologies to be applied are decided upon. To make architecture exploration feasible, a system-level specification should not be biased towards a particular choice of implementation and should be described in a language that abstracts from technology. To help designers bridge the gap between requirements capture and synthesis, adequate tool support is indispensable. Tools are required to edit, validate, verify, simulate and transform specifications, but also for parameter estimation, automatic test-suite generation and system-level synthesis to compile specifications to technology-specific languages such as VHDL and C. To support the development of such tools, a system-level language should be equipped with a mathematical semantics [4]. To support verification and estimation, the language further should be able to express several constraints such as functional, timing, architectural and technological constraints.

From the characteristics of reactive systems, we presented in Section I, a number of other language requirements can be deduced. To cope with complexity, a system-level language must have powerful abstraction mechanisms: design entities with well-defined interfaces, encapsulation, hierarchy, data abstraction and procedure abstraction [4]. Further adequate concurrency, synchronisation, communication and real-time primitives are needed. To make a system-level language suitable for architecture design, architecture structure and topology should be expressible.

III. BASIC CONCEPTS OF POOSL

In [5], [4] we introduced the method Software/Hardware Engineering (SHE). SHE is an object-oriented modelling technique. Starting from informal graphical models, SHE produces rigorous behaviour and architecture models expressed in the formal specification language POOSL [4]. The POOSL language is specifically developed to be used as a system-level design language. It satisfies many (but not yet all) requirements stated in the previous section.

The POOSL language combines a process part with a data part. The process part is based on the key ideas of the process algebra CCS [6], [7]. The data part is based upon the concepts of traditional sequential object-oriented programming languages such as Smalltalk [8], C++ [9] and Eiffel [10]. In POOSL very complex reactive real-time behaviour is represented by a collection of asynchronous concurrent process objects that communicate synchronously or asynchronously by passing messages over static channels. Behaviour of process objects is described by

- synchronous (conditional) message (and data object) passing primitives;
- asynchronous broadcast primitives;
- choice and select primitives, guarded commands, parallel composition, procedure (method) abstraction and (tail) recursion;
- interrupt, abort and delay primitives.

Next to processes, POOSL supports the concept of
cluster. A cluster is composed of processes and other clusters and acts as an abstraction of these. Clusters are used to create an hierarchical structure of modules that hide their internal structure. The constituents of a cluster are composed by parallel composition, channel hiding and channel renaming. These combinators are based upon similar combinators originally used in CCS [6], [7]. Together, clusters and channels are suitable for describing architecture structure, topology and implementation boundaries [4].

To describe complex functional behaviour, POOSL supports data objects. Data objects have a sequential behaviour and communicate by synchronous message passing. They are contained in processes and they model the private (non-shared) data of these processes. Data objects are also called travelling objects, since they can be passed between processes.

The POOSL language is equipped with a complete mathematical semantics. The semantics of the non real-time part of POOSL is given in [4]. The formalisation of the real-time extension is described in [11], [12].

The key feature of POOSL is the expressive power to model very complex dynamic (communication and functional) behaviour as well as static (architecture and topology) structure in an object-oriented fashion. The language can best be compared with the formal description languages LOTOS [13], [14] and SDL [15], [13]. However, there is a very important difference. The data parts of LOTOS and SDL are based on abstract data-typing languages. These languages are attractive from a mathematical point of view, but they have appeared to be difficult to understand [16] and they are hard to implement efficiently. This is one of the major reasons why we decided to base the data part of POOSL on the concepts of object-oriented programming languages. This decision resulted in a language that is more expressive, easier to understand and easier to implement than LOTOS and SDL.

IV. Example

In [17] we have shown the suitability of the SHE method and POOSL language for the specification and design of an industrial distributed control system. In this section we will demonstrate how the POOSL language can be used to model a non-trivial part of a protocol stack [18]. The protocol stack is shown in the simulator window of Figure 2. The stack consists of two peer NetworkLayers, two peer DataLinkLayers and a PhysicalLayer. These entities communicate by exchanging messages over channels. The peer Net-

workLayers exchange Packets. Each NetworkLayer can send messages of the form packet(somePacket) to its corresponding DataLinkLayer by executing NL-toDL!packet(somePacket) statements. toDL is a port of the NetworkLayer object and this port is connected to channel NLtoDL1. The DataLinkLayer can receive the packet(somePacket) messages by executing fromNL?packet(somePacket) statements. Here somePacket denotes a data object of data class Packet. Upon reception of a Packet the DataLinkLayer wraps this Packet (together with some control information) in a data object of class Frame. This Frame is then sent to the PhysicalLayer which transports it to the peer DataLinkLayer. Consequently, the Packet is retrieved from the Frame object, and is delivered to the peer NetworkLayer.

Fig. 2. A Protocol Stack

Fig. 3. PhysicalLayer cluster

The PhysicalLayer is unreliable and can loose messages. To make sure that all Packets are delivered in the correct order, the DataLinkLayers make use of a sliding-window protocol [18]. Therefore, DataLinkLayers can send some Frames several times and they can also send (piggy-backed) acknowledgement Frames.
The specifications of the DataLinkLayers are not further elaborated in this paper. Instead we will take a closer look at the specification of the PhysicalLayer.

PhysicalLayer is a cluster consisting of two SimplexChannels, see Figure 3. A SimplexChannel has two ports in and out. From port in it receives frame(aFrame) messages and after some specified period of time it delivers these messages at port out. The specification of the behaviour of a SimplexChannel is shown in the Process Class Browser of Figure 4. Within the Instantiation Parameters pane, two instantiation parameters, errorDistribution and transTimeDistribution, are specified. These parameters are bound at the moment a process of class SimplexChannel is instantiated. Parameter errorDistribution is a data object of data class Bernoulli and it models the probability that a message in the channel is lost. The parameter is bound to expression new(Bernoulli) withParam(1.0) which delivers a new Bernoulli distribution with success parameter 1.0 (no messages are lost), see Figure 3. The transTimeDistribution parameter models the transmission time of a message through the channel. This parameter is bound to a Normal distribution object with mean 10.0 and variance 0.0 (the transmission time is constant).

The behaviour of a process is specified in terms of instance methods. Methods can be compared to procedures of traditional imperative programming languages such as C or Pascal. Upon instantiation of a process, it calls its initial method. The initial method of class SimplexChannel is transferFrames(). The definition of this method is given in the Edit Method (POOSL) pane of Figure 4. The definition starts with header transferFrames()(). The brackets indicate that methods can have input and output parameters. Clause | aFrame:Frame | declares a local method variable aFrame of data class Frame. The declaration of local variables is followed by the actual body of the method. Message-receive statement in?frame(aFrame) indicates that the process wants to receive (?) message frame with parameter aFrame from channel in. This statement is blocking; it is only executed if some other process executes a corresponding message-send (!) statement of the form in!frame(someFrame).

After the reception of message frame(aFrame) a SimplexChannel executes two statements in parallel. Parallel composition is indicated by the par · · · and · · · rap construct. The first statement of the parallel composition starts by drawing a sample from the Bernoulli distribution and by checking whether this sample denotes success. This operation is performed by sending the message yieldsSuccess to data object errorDistribution. If this data object replies with false the message is lost (statement skip is executed). Otherwise, the SimplexChannel draws a sample from the Normal distribution transTimeDistribution by sending it the message next. Consequently, the channel starts to delay for the sample’s amount of time units. After this delay, message-send statement out!*frame(aFrame) is executed. The * symbol denotes that the message is sent asynchronously, i.e. the message is lost if no process is currently willing to receive it. The second statement transferFrames()() of the parallel composition is

Fig. 4. SimplexChannel specification
a method call. This statement starts the method, that is currently being executed again in parallel. In this way, the channel is able to transport an unbounded number of incoming messages.

Notice that the behaviour of the SimplexChannel is far from trivial. Due to the expressive power of POOSL however, the behaviour can be expressed in only a few lines of easily readable code. This expressive power makes POOSL especially suitable for describing (the behaviour of) complex systems at a high level of abstraction without having to write detailed implementations in some hardware or software description languages such as VHDL or C (the reader is challenged to describe the behaviour of the channel in one of these languages).

V. TOOL SUPPORT

The drawings we used in the previous section are snapshots taken from an interactive model editing and simulation tool for the SHE method. The tool is used to incrementally specify and modify classes of data, processes and clusters. A specification does not have to be complete before it can be tested and simulated. Using the different buttons on the Drawing window of Figure 2, the current (partial) model can be executed in different modes of simulation. The messages and parameters that are passed between the different processes and clusters are indicated on the appropriate channels. It is possible to open inspectors on each model part (data objects, process objects and clusters). The inspectors show the current state of each variable and for processes they show the code that is currently being executed as well.

To inspect the history of all messages exchanged between the different entities, interaction diagrams (also called message sequence charts) can be generated automatically. An example of an interaction diagram is shown in Figure 5. The diagram shows the different model entities and the messages exchanged between these entities. For each message, the time, the channel and the parameters are indicated. For instance on time 15, message packet with parameter 1 is exchanged between the left-most NetworkLayer and DatalinkLayer. Notice that not every instance of message exchange is shown in Figure 5. The communications inside the PhysicalLayer are left hidden. They can however be made visible by either exploding or entering the PhysicalLayer, see Figure 5. Another way to handle the complexity of complex behaviour is through the concept of scenario [4], [19]. A scenario defines a coherent piece of behaviour in terms of the entities (processes, clusters and channels) that are involved in this behaviour. The interaction diagram of Figure 5 visualises only the behaviour corresponding to the NL1toNL2 scenario. This scenario involves all processes and clusters shown in Figure 2 as well as channels NLtoDL1, DLtoPL1, PLtoDL2 and DLtoNL2.

Next to the development of an advanced simulation and validation tool, research is carried out into system-level synthesis of POOSL specifications. The first results into the mapping of POOSL onto hardware are presented in [20]. For the automatic mapping onto software (C++), promising results have been obtained. Further a system of behaviour-preserving transformations has been developed [4], [21]. Behaviour-preserving transformations allow the architecture of POOSL models to be modified, without changing the functional and communication behaviour. The transformations are not yet supported by our simulation environment.

VI. CONCLUSIONS AND FUTURE RESEARCH

Industry is currently facing a crisis in the design of complex systems. They are not able to simulate, validate or verify the current generation of systems. The EDA industry is trying to define a new standard language, called System Level Design Language (SLDL), that is to be used at higher levels of abstraction, before hardware/software partitioning and technology choices have been made.

In this paper we propose a new system-level modelling, specification and design language POOSL (Parallel Object-Oriented Specification Language). POOSL is developed as part of the SHE (Software/Hardware Engineering) method for reactive hardware/software systems. The language is able to express very complex real-time communication and functional behaviour in the form of process objects and data objects. It can also represent the architecture structure and the topology of systems in terms of clusters and channels.

A number of supporting software tools have been developed. An available interactive simulator tool allows POOSL models to be entered, simulated and validated. Model validation is supported by message sequence charts and scenarios. System-level synthesis for software and hardware are being investigated. The first results for both software and hardware synthesis have been obtained. To support architecture design, a system of mathematically proven behaviour-preserving transformations is developed. Important
future research topics involve the automatic verification of (real-time) properties and the estimation of system parameters supporting hardware/software partitioning.

REFERENCES