Accepted for publication in *Proceedings of VHDL International User's Forum Spring '98 Conference.*

Considerations on System-Level Behavioural and Structural Modeling Extensions to VHDL

Peter J. Ashenden
Dept. Computer Science
University of Adelaide, SA 5005
Australia
petera@cs.adelaide.edu.au

Philip A. Wilsey
Dept. ECECS, PO Box 210030
University of Cincinnati
Cincinnati, OH 45221-0030, USA
phil.wilsey@uc.edu

Abstract

This paper reviews the requirements on a language for modeling behaviour and structure at the system level, and considers possible approaches to extending VHDL to meet these requirements. Modeling issues that obtain in a system-level design language are identified, including abstraction of data, concurrency, communication and timing, and design refinement. Some system-level design languages and notations are surveyed, and previous proposals to extend VHDL for system-level design are reviewed. Specific language design issues for extending VHDL are discussed, and some alternative solutions are presented.

1. Introduction

One of the main challenges facing designers of complex integrated systems is that of managing complexity. Many systems are now composed of complex digital and analog hardware, managed by embedded software. In some cases, they must communicate with other systems, forming part of a larger distributed system.

The key to managing complexity in such systems is the use of abstraction. Designers focus first on the abstract properties of a system in various domains, and devise a systems architecture that will satisfy the requirements placed on the system. They may devise multiple architectures, analyze them, and choose among them based on the analyses. The domains under consideration include behaviour, structure, performance, physical arrangement and packaging, power consumption, thermal, cost, etc. In each domain, abstraction is used to focus on the major aspects of the system, and minor detail is ignored. Judicious choice of abstractions makes architectural design and analysis tractable, and aids subsequent partitioning and refinement of the system design.

A number of specialized languages may be used in designing a system. Some, such as the Unified Modeling Language (UML) [17, 35], are primarily notations for expressing the relationships between the various components that make up a system. Others, including those discussed in this paper, focus on expressing the required behaviour and logical organization of a system and its components. Once behaviour has been captured using such languages, the system can be simulated to verify that it meets its requirements. Subsequently, it can be manually or automatically refined to a more detailed implementation in terms of hardware, software, or a combination of both.

Hardware description languages also focus on describing systems in the behavioural and structural domains. However, due to their origin as languages for hardware design, they do not include strong capabilities for abstracting over data and for describing complex interactions. For example, in Verilog [26, 39], data types are closely bound to their binary representation, and signalling between modules includes aspects of electrical implementation. VHDL [2, 25], on the other hand, allows more abstract expression of data, since its type system is similar to that of conventional programming languages. However, its signalling features are still closely bound to electrical implementation.

This paper reviews the requirements on a language for modeling behaviour and structure at the system level, and considers possible approaches to extending VHDL to meet these requirements. Section 2 reviews the modeling issues that obtain in a system-level design language. Section 3 briefly surveys system-level design languages and notations that are widely used, and Section 4 surveys previous proposals to extend VHDL for system-level design. Section 5 focuses on specific language design issues for ex-

---

* This work was partially supported by Wright Laboratory under USAF contract F33615-95-C-1638.

tending VHDL, and discusses some alternative solutions. Section 6 summarizes and presents conclusions.

2. Modeling Issues for System-Level Design

At the system level, a system can be modeled as a collection of active objects that react to events, including communication of data between objects and stimuli from the enclosing environment. Abstraction is needed in a number of areas to make system-level behavioural modeling tractable:

- abstraction of data,
- abstraction of concurrency, and
- abstraction of communication and timing.

It is important to note that the issues of concurrency and communication have been widely discussed in the literature on concurrent programming languages. (See Ball *et al* [7] for a survey.) We draw on experience from that field in considering the design of system-level modeling languages.

Early in the design flow, the representation of data in a system should be at a high level of abstraction. For example, data may initially be modeled using uninterpreted tokens, allowing performance of a design to be estimated based on token flow rates and queuing delays. When data types are refined to concrete representations, abstract data types and object-oriented techniques [10] can be used to manage the increased complexity. Refinement to hardware-specific representations should be deferred to later in the design flow. Thus, a requirement for a system-level design language is a facility for expressing uninterpreted and abstract data types.

A system-level design language needs to allow expression of concurrent processes representing the active objects in a system. In some systems, the number of active objects is not statically determined, but may vary during operation of the system. For example, in a client/server system, new service agents may be created as requests arrive from clients, allowing multiple requests to be processed concurrently. In order to describe such systems, a system-level design language must allow expression of process types that may be dynamically instantiated and terminated.

At the system level, processes representing active objects must interact to communicate data and to synchronize their operation. The simplest form of interaction is message passing, involving the transfer of data from a sender process to a receiver. The act of message passing can also be used to synchronize the processes. We focus on message passing in this paper, as it a natural abstraction of communication common to both software and hardware. Other forms of interaction, such as rendezvous and remote procedure call are possible [7], but are oriented specifically toward software implementation. They can, however, be expressed in terms of message passing.

There are two ways in which message passing abstracts away from the details of communication in hardware description languages. First, communication events are not tied to specific times, but rather are simply ordered by relative time of sending. This causality-based ordering is weaker and less constraining than clock-time ordering, and so is more appropriate at the early stages of design. Second, communication events may be queued (either by queuing messages or processes), rather than relying on the recipient sensing data at the correct time. This allows multiple communication events to form a stream or a transaction without the need for detailed signalling protocols.

System-level design occurs at the early part of a process that involves refinement to lower-level hardware and software implementations [18]. A model at the system level should be expressed in a form that enables verification that a refinement correctly implements the model. Possible approaches include behavioural synthesis [32] (correct by construction), and formal verification using model checking and equivalence checking [23, 33]. Refinement to a software implementation is facilitated by a system-level modeling language that is closely related to programming languages. In principle, the hardware and software implementations could be expressed in the same language as the system-level model, thus avoiding semantic mismatches between different languages in the design flow.

Given a system partitioned at the architectural level, it is important to specify the interfaces between the modules. Incorrect implementation of interfaces is a significant source of defects in system designs. Correct implementation is aided by explicitly modeling the interfaces, including the data representations used and the communication protocols. Explicitly modeled interfaces can also be used to aid refinement of the system, by describing transformations between higher-level and lower-level representations and protocols.

3. Review of System-Level Design Languages

There are several languages for system-level specification and design that are widely used. A number of them, such as Z [15, 36] and LOTOS [9, 28], focus on specifying a system in a declarative style. Others adopt a process-oriented approach to describing the structure and behaviour of a system at a high level of abstraction. In this section, we review four languages in the latter category, showing how they do, or do not, address the requirements discussed in Section 2.

3.1 StateCharts

StateCharts [20] is a graphical notation for expressing the behaviour of a system. Its abstraction of concurrency and communication is based on a extended finite state machine (FSM) model. Extensions include hierarchical states, in which a superstate represents a subgraph of the FSM; the machine is in the superstate when it is in any of the substates. StateCharts also allows concurrent composition of states; the machine is in both of the states. If the concurrently composed states are hierarchical, those states act as concurrent submachines. The global machine is in some substate of each of the concurrently composed states. Thus, each concurrently composed state can be viewed as a concurrent process in the model. In StateCharts, these processes are statically specified using a graphical editor. The effect of a StateChart on its environment is specified by *actions* associated with states and transitions. Actions include starting and stopping *activities*, such as generating outputs on signals connected to the environment.

Transitions in a StateCharts FSM occur are triggered by *events*, and may optionally be guarded. An event may be input from the environment, a transition in a concurrently composed submachine, or a time-out. Guard expressions may be be predicates over the current state of concurrently composed submachines. As Harel comments [20, page 264], the use of the state of one submachine to cause events or guard transitions in other submachines implies a form of broadcast communication between the processes representing submachines. There is no explicit timing involved in the StateCharts notation beyond the causal ordering of transitions fired by events, with actions being performed instantaneously.

While the StateCharts notation addresses the issues of abstraction of concurrency, communication and timing, albeit with a somewhat limited set of abstractions, it does not adequately address abstraction of data. Using events for communication limits StateCharts to uninterpreted modeling of data. Thus, the notation is mainly useful for modeling the control-oriented aspects of a system. The notation also does not deal with modeling of the interfaces between components. Indeed, it violates the principles of modularity and information hiding (see Booch [10]) by allowing state information nested inside of one submachine to be names in event and guard expressions in other submachines.

The StateCharts notation has been adopted in the Uniform Modeling Language (UML) [17, 35]. In that version of the notation, events can be parameterized, allowing communication of data between processes via events; states can have variables; and actions can be procedural, allowing update of variables and sending of messages to objects.

3.2 Estelle

Estelle [11, 29] is an ISO standard language for describing distributed information processing systems. The concurrency aspects of a system are described using a hierarchy of instances of *modules*, each of which is a concurrent process. A module has *interaction points* for communication, and may have locally declared variables and nested module instances. The behaviour of a module is specified in terms of an extended non-deterministic state machine, with Pascal-like statements included as transition actions. Transition conditions are expressed in terms of message arrival at an interaction point, guarded by boolean expressions. Modules may be statically instantiated, or may be dynamically instantiated as part of transition actions. The modules within a group of processes called a *subsystem* execute with synchronous parallelism; first, enabled transitions are selected, then the selected transitions are executed. Modules in different subsystems are permitted to execute with asynchronous parallelism.

Communication in an Estelle description takes place using buffered asynchronous message passing over typed links between interaction points. The links may be statically created, or dynamically created as part of transition actions. Estelle provides for abstraction of interfaces in the form of *channels*, which specify a protocol for message exchange over linked interaction points. A channel defines a set of roles, and, for each role, a set of allowed messages that may be communicated. An interaction point of a module may be declared to take on a particular role of a declared channel type. While Estelle does provide richer facilities for expressing data than StateCharts, they do not extend to expression of abstract data types.

3.3 SDL

SDL [16, 31], like StateCharts, is a language for describing the behaviour and structure of systems. The language has both textual and graphical notations, with the same underlying semantics. A system is described as a statically specified hierarchy of *process sets*. Within each set, processes may be either statically or dynamically created instances of declared *process types*. Process sets are connected with statically specified typed channels (*signal routes*), over which the processes may communicate using buffered asynchronous message passing. A process may have locally declared variables. The behaviour of a process is expressed in terms of an extended finite state machine model. Transitions are enabled by message arrival, and possible actions on transitions include variable assignment, process creation, sending a message, and a form of remote procedure call.

SDL has sophisticated features for abstracting over data types, based on the ACT-ONE model for defining abstract data types (ADTs) using axiomatic semantics [13]. A number of types are predefined, including numeric, boolean, character and time types. The model also allows specification of composite types, such as record, array and list types.

The revision of the SDL standard in 1992 enhanced the language to include object-oriented features. For example, process types may be specialized through inheritance; the specialized process type inherits the variables, states and transitions of the parent, and may add new variables, states or transitions. Data types and signal types may also be specialized through inheritance, with new elements being added to the specialized types.

3.4 CSP

Hoare initially proposed Communicating Sequential Processes (CSP) as a programming language for concurrent systems [21], and subsequently developed the notation into a formal mathematical theory of communicating concurrent systems [22]. In the programming language form, a system is described as a set of statically specified processes that commu