SPECIFICATION DRIVEN BEHAVIORAL DESIGN OF COMPLEX SYSTEMS

Oryal Tanir
Director - Business Engineering and Simulation,
Bell Canada
Montreal, QC, Canada

Traditionally in the design automation field, simulation has been applied to explore low-level design details. However, a focus upon design specification driven simulation environments has been missing. Such tools can conceptually capture an initial design specification and allow design exploration at a highly abstract level with a simulation-based environment prior to synthesis. At this design stage, software and hardware elements can be indistinguishable and represent a good opportunity for addressing system level concerns such as partitioning, co-design and architectural trade-offs. With the advent of more complex systems, the ability to model and verify properties of various alternative designs is mandatory to produce cost effective and sound systems.

Specification driven design implies a need to support architectural design and rapid prototyping systems (RPS) within a design flow. The system design activity is generally the starting point within the design phase of a product life-cycle - which involves the design capture of specifications into an executable model. Hence language requirements at this stage encompass modeling capabilities. The design and modeling of the conceptual system at this abstract level implies that the design environment must support concepts such as generic model reuse, component and structural reuse, intelligent library management, and hierarchical design. After a suitable model is defined, the language must provide support for experimentation and analysis. These activities are crucial for a designer to explore a given design space, make appropriate trade-offs and partition the design to different hardware/software configurations. Such activities can be supported through design simulators and formal methods.

This paper examines the benefits of applying a specification driven approach and presents a framework for environments that can support the related design activities. The Design Analysis and Simulation Environment (DASE) based upon this framework has been successfully implemented through a joint initiative between Bell Canada and McGill University.

1. Introduction

Specification driven environments can conceptually capture an initial design specification and allow design exploration with a simulation based environment prior to synthesis. With the advent of more complex systems, the ability to model and verify architectural properties of various alternative designs is mandatory to produce cost effective and sound systems.
The progression of tool capabilities extending toward initial specifications promises to reduce design ambiguities and errors early in the product life cycle. Such tools must be capable of design capture at increased levels of abstraction to support modeling at the architectural design level. The architectural level is the most abstract design level defined in Bell and Newell (1971) where the behavior of system elements are viewed as communicating which may describe constructs such as processors, memories, switches or other high level descriptors.

With the increased level of abstraction, the design space for a “general purpose” architectural level RSP tool is heavily heterogeneous making it unfeasible to derive a model from most initial specifications. As a result, it is generally agreed that a successful architectural level RSP should be domain specific (exhibiting a reasonably homogeneous design space) or support a wide variety of specification paradigms. The role of modeling and simulation in many instances is across many groups for a given design. However, the information and knowledge gained with the activities are more likely to be retained only by the individual groups. The models are generally not portable or re-usable between groups. For example, a hardware model is rarely used by software designers which implies that potential design trade-offs must either be performed before or after the individual design. In the architectural level design, the system is designed at a high level of abstraction involving both software and hardware elements. Models can be re-usable in the sense that past designs can be incorporated into new ones permitting more design alternative explorations. Domain knowledge is also used as input to help direct the outcome of the design.

This paper examines the benefits of applying a specification driven approach and presents a framework for environments that can support the related design activities. The paper also outlines interrelated advanced topics such as intermediate architecture languages, model libraries and model verification issues.

2. The cost of Errors

A typical design flow for large system design is represented in figure 1. The figure highlights the high level flow of information from design concept to implementation. Four phases of a product’s early life cycle are also shown.

- The specification phase is where the requirements and behavior of a conceptual system are first formulated. The result of the phase is a document and/or model reflecting the desired characteristics of the system.
- The design phase is where different solutions are explored and evaluated to meet the desired characteristics of the system. Typically, this will involve the creation and simulation of different models of the system. The eventual goal is to produce a description of the design which complies with the specifications. Ideally, the description will be executable, so that it is automatically used within the next phase.
- The implementation phase constitutes all the activities that contribute to the physical manifestation of the design. The result is the first version of the product. This is generally an intermediate release of the final product.
- The test phase covers all activities to validate the product’s behavior against the specification. The product is then ready to be released to the customer for acceptance testing.

Within each phase there are activities related to the verification of the design. Design errors can therefore be uncovered during each phase. A cost curve in Figure 1 indicates the relative cost to fix design errors uncovered in the respective phase. As can be seen the cost increases by orders of magnitude from phase to phase. It is highly desirable to uncover most of the design errors as early as possible. In
particular, modeling activities within the specification and design phases (or specification driven design) can help reduce major system errors from creeping up in activities later down the product’s life-cycle.

A key ingredient for the success of designing complex systems is the capability of an environment to model and represent a designer’s conceptual system in some computer readable form. The model must be at a sufficiently abstract level to permit the designer to concentrate on relevant details and not wallow in unnecessary detail. The emphasis of the designer must be more on model manipulation and exploration of solutions to the problem rather than the process of creating the model.

3. Specification Driven Design

All designs will usually originate from a set of specifications or requirements. For purposes of this paper, requirements are system characteristics driven by the customer or the marketplace; and specifications are a technical mapping of the requirements.

Ideally, the specifications will be defined formally and remain unchanged until the construction of the desired product. In reality this is almost never the case. Requirements or specifications tend to change during the course of time by unforeseen factors. All attributes of the desired system are not well-defined to generally articulate it in a formal manner.

Specification driven design can be viewed as a transformation function where it accepts design specifications, design requirements and design constraints as input. Through a range of activities, the output of the function is to produce a hardware / software model of a design that is consistent with the criteria presented at the inputs. The nature of architectural design implies a hierarchical approach to the design of system, progressing from a high-level system detail to a low-level description.

Given the desired level of abstraction that designers will work with, specification driven modeling environments need to perform three fundamental activities: system design, model execution and design synthesis (Tanir, 1997). These can form the basis of a framework as shown in Figure 2.

4. Intermediate Representation

An essential ingredient to realize an architectural design environment is the use of an internal medium for capturing the input. This is depicted as the high-level, intermediate and synthesis models in the figure. The language (or group of languages), which will provisionally be referred to as an intermediate language, needs to accommodate both the formal and informal representation (Biddle, 1996). This is not a simple task. System designers have typically utilized many different languages to capture relevant aspects of a system. It is also generally agreed that one unified language or methodology is not capable of representing systems on all levels of design abstraction (Ramming, 1993). It is also observed in the milieu that as the
abstraction level increases, the analysis methods used in tools shift from a simulation oriented one to a formally based one. Hence it is difficult to use an existing paradigm to address design concerns across a broad range of activities at the architectural level of design (Monroe, 1996).

All this implies that the intermediate representation supported by the environment must undergo some transformation during the design activities to alter the vague semantics into less ambiguous ones. This transformation could be made possible with some constructive interjection by the designer and resources allocated to the environment.

The basic requirements for an intermediate language to support architectural design are:

• Must be able to capture formal specifications. This implies that the language should bear formal foundations, or a direct translation from the formal specification to a subset of the language should be possible.

• Must be able to capture informal and partial specifications. This implies that the language can permit the representation of incomplete structures such as missing interface descriptions.

• Must support high-level communication paradigms (typical of software) such as message passing.

• Must be able to represent low level communication entities typically found in hardware description languages.

• Must be able to distinguish between software and hardware elements.

• Must provide for the dynamic alteration of a model’s structure and behavior as required by the designer or environment. This implies that the interface descriptions of the models can be altered on-the-fly at run-time.

5. System Design

The system design activity is generally the starting point within the design phase of a product life cycle. The activity involves the design capture of specifications into a model that can then be executed.
Hence, the language requirements at this stage encompass modeling capabilities. A model represents an abstraction of a domain under study. In the context of architectural level design, an abstraction is closely linked to specification capture, re-usability, and object-oriented features. However, from the model perspective, it can be seen as representing a set of software or hardware functionality. This transparency lies in the fact that the behavior as expressed by the model can be viewed as a set of communicating processes or states. Hence at this high level of description there is no direct correspondence to the underlying hardware or software nuances.

The output of the system design activity is an internal representation that reflects the behavior and structure imposed by the inputs. To facilitate addressing the different input possibilities, most of the formalisms that are applicable as the internal representation language may need object-oriented capabilities or extensions. This is to minimize the large amount of complexity in transactions between the diverse model constructs. It should also be noted that models defined in the internal representation can also be used as an input if so desired (especially if the internal representation is a well-defined formalism).

6. Model Execution

After a suitable model is defined, the language must provide support for experimentation and analysis. The activities encompassed by experimentation and analysis will be referred to as model execution. These are crucial aids for a designer to explore a given design space, make appropriate trade-offs and partition the design to different hardware/software configurations. Such activities can be supported through design simulators and formal methods.

A derived or transformed model is achieved through multiple intermediate refinements, which are a consequence of the interaction of the analysis, execution engine and decision entities of the environment. User controlled incremental enhancements to the model eventually converge to a detailed model which can be used by automated design aids to generate hardware and software components of the system. The resulting design is not necessarily optimal (a very difficult problem in itself) but a solution within the given set of specifications. As the execution engine moves the model through simulated time, decision support tools are needed to complete the successful execution of the model.

A decision support system implies the integration of two components to the design environment: a library support system and a rule based engine.

The library support system provides many essential elements to the designer. It can facilitate the configuration of the executable model by providing re-usable models from a model base. Hence powerful storage, retrieval and management features of the library are desirable. The library should also maintain a class of models. This permits the environment to interject during execution errors resulting from inadequate model choices. Such conditions can typically occur if the modeled sub-component does not perform to within the initial specifications.

Reuse is usually though of as utilizing behavioral elements of a past design and reapplying them in a new one. However, structural aspects of the design can also be exploited. Systems that support such a methodology can facilitate the construction of complex systems that exhibit some symmetric properties (Tanir and Erdogmus, 1997a).

A rule-based engine is needed to permit the environment to compare the system under design with past design knowledge. A rule support system can also facilitate the configuration of models extracted from a model library. For example, if a given library model can indicate how it needs to be structurally and behaviorally created within a given system model, the process can be automated by the system - permitting the construction of generic models.
7. Synthesis

Model execution activities are repeated, refining the model until a final model is derived which meets the designer’s expectations and which can also be synthesized. In general, synthesis implies taking a set of behaviors, constraints and goals and generate a suitable structure that can implement the behavior while satisfying the constraints and goals.

Any environment working at the architectural level must support design partitioning, timing transformations, and translation of behavior.

Traditional synthesis techniques and tools, which are applied at lower levels of design abstraction, concentrate on either hardware or software. Partitioning at these lower levels of abstraction is concerned more with resource allocation and data flow optimization. At the architectural level, partitioning is significant and implies the identification of the software and hardware components since they are initially indistinguishable. This is not a trivial task and requires some heuristics and optimization criteria.

To support translation, the environment must provide an added capability, traditionally not part of a simulation or synthesis environment. This is the capacity to record and store design changes made during the design execution phase. The records must provide the behavioral information as well as structural modifications. Conventional simulation environments have provisions for observing the execution and capturing statistical information. However, the underlying simulation model in these environments are static both in structure and behavior. In the architectural design world, the model is viewed as a dynamic entity - changing its behavior and structure during the course of design execution. In this sense, an architectural level model assumes a design history which reflects the different roles it has played.

8. DASE

The paper has presented the fundamental criteria for specification driven design. To illustrate how these can be applied, this section presents a specification driven environment for telecommunications systems called DASE (Design Analysis and Synthesis Environment). DASE utilizes an internal specification language called the Design and Specification Language (DSL) (Tanir et al. 1995a). Design specifications are entered and captured by the environment using the internal architectural “meta-language” DSL and supported by an object oriented library system. The language is interpreted through a DSL interpreter which also interacts with a DSL simulator. The various components of DASE are shown in Figure 3.

Specifications represented as DSL models comprise of message passing object-oriented entities called modules. These are abstract model building blocks that possess behavior(s), hierarchy (higher-order modules), inheritance (generic modules), constraints and internal data entities (resources) used to store local variables, registers or state information. The behavior of a module represents the actions undertaken when events occur. An event occurs upon the arrival of a message to a module. The module then attempts to execute the behavior associated with the message. Messages are internally queued and processed sequentially.

Details of DSL and the related design environment can be found in Tanir (1997). Hence, only a quick overview of the language is presented in this paper.

Module behavior is a procedural description of actions which, within satisfaction of a set of constraints, can consist of:

i). Communication initiation with other modules.
ii). Modification of resources or data structures associated with the module, and
iii). A suspension of further operations for a specified time period.
Modules are “interconnected” through the definition of higher order modules. These are modules that contain structural information. Connections are described using “ports”. A port is a virtual communication channel between the module and its environment. No direction need be specified to the port. The environment deduces direction from the information flow across ports. Port specifications are not typed so that different levels of abstracted information may flow through the same port.

Object-oriented Library support features allow for the creation, storage and retrieval of module libraries in an organized manner. Libraries maintain all the information related to a module. These include information regarding any constraints to be imposed on the modules, any configuration rules to be applied to the components of the library, and the interface specification of the library module. The interface specification is created by the library system to define exactly what ports are available for communication with the library module. The language also allows system and module level constraints to be defined.

Constraints define limits upon the structure and behavior of modules. They can be classified as local (dependent upon parameters from one module) and system (dependent upon more than one module) constraints. The former parameters are known beforehand whilst the latter is configuration dependent. For example, the maximum size of a memory module, the minimum delay period for message reception, or the maximum number of calls possible on a switch are represented as local constraints. The size of a cache module, determined through a calculation of the number of processor modules in a multi-processor model is an example of a system constraint and is calculated upon invocation of the whole DSL model.

The simulator uses a DSL model to configure and setup the relevant constraints and resources to support DSL simulation. During simulation, the simulator interacts with the user upon detecting a constraint or simulation violation. The simulator searches for alternate design modules from an existing model base which belong to the same module class and rerun the simulation. The user can also relax the problematic constraint and proceed with the simulation.

When the designer is satisfied with the simulation results, the synthesis stage may be initiated. An analysis methodology also exists whereby the DSL modules can be translated into predicate/transition petri-nets. A synthesis tool within the DASE environment translates the DSL constructs into concurrent entities in VHDL.
This section illustrates a typical progression of activities that may occur in a design scenario using DASE. The example is that of an Asynchronous Transfer Mode (ATM) switch design. Due to space limitations, significant behavioral details of the model are not included in the paper.

A generic representation of an ATM switch is shown in Figure 4 (Tanir et al., 1995b). Within DSL, this is described by the statement:

\[
\text{Module}(\text{switch}(M), [\text{sf}(M), \text{in}(M), \text{out}(M)]).
\]

The parameter “M” defines the size of the switch (number of input or output ports). It is used by the module in configuring its sub-components. Many different components can be generically defined and reused through the library. The design specification may originally dictate some of the design elements. Alternatively, the more detailed design decisions can be made during the model execution.

A typical design choice in this example can be the switching fabric topology (shown as the “SF” module in the figure). The switching fabric (a central component to most switch designs) can be designed in many different ways. Two such architectures are the knock-out and banyan designs. This example can also be approached using a structural reuse/design point of view. For further information, the reader is referred to Tanir and Erdogmus (1997b).

The Knockout switch fabric is shown in Figure 5. It requires some synchronous control hence a clock message is necessary to harmonize the transfer of data internally within the fabric. The Banyan switch fabric (Figure 6) is an asynchronous design, allowing data to flow through as needed. The basic principles of the designs are not relevant here, but can be found in Awdeh et al., (1995).

The ATM switch design using DASE would commence with a default choice for the switch fabric – which is the Banyan topology. Hence, unless otherwise specified by the designer, the Banyan topology
will be instantiated as the switching fabric. The actual instantiation of the relevant modules and their configuration is automatic. An algorithm to generate the interconnections for the modules is given below:

Stage = \log_2(M);
k = M/4;
For S = 0 to (Stage-1) do
Begin{
    For L=0 to (2S - 1) do
    Begin{
        For ii=0 to (M/(2x2S) -1) do
        Begin{
            i = ii+Lxk;
            j = floor((ii/2)) + 2xL;
            create a path: path( node(S, i), node(S+1, j), [ out(0), in(y) ]);
            create a path: path( node(S, i), node(S+1, k+j), [ out(1), in(y) ]);
        }
    }
    k = k/2;
} end (main loop)

The algorithm will generate the interconnections for any size switching fabric. The modules assume that the output ports are named out(0) and out(1) whereas the inputs are in(y). The labeling for the inputs are not necessary and can be omitted unless the module behavior specifically requires the input port labels. The algorithm requires one parameter (the switch size M) to create all the module interconnections. After instantiating the nodes, the algorithm is used to establish the interconnections. Such algorithms are part of configuration rules within the DASE library mechanism. They are stored and associated with respective library components.

Fig. 5 The Knockout Switch.
Once the appropriate sized ATM switch fabric is configured, the entire switch is instantiated in a similar fashion. Design execution then takes place. If a constraint associated with the switch fabric is violated (such as latency, delay or buffer size), the DASE simulator will access the library for an equivalent replacement (in this case a knockout switch would be selected). The user will then be given the option of replacing the Banyan fabric for the Knockout one, or relaxing one of the constraints associated with the fabric.

The design execution progresses along these lines until a suitable design that can be synthesized is achieved.

10. Conclusions

In summary, a specification driven approach applied to complex designs can have a significant impact upon the design cycle. Some of the critical elements a supporting framework must possess are: good library support, a powerful intermediate representation for its models, and guidance for design exploration and synthesis support. An environment (DASE) to support specification driven design based on component reuse has been presented. Further research has also been undertaken in collaboration with the National research Council of Canada in creating a structural reuse front-end interface with DASE.

11. Acknowledgments

The DASE work has been a result of a long collaboration with Professors P.C.P. Bhatt and V. Agarwal who were (at the time) at McGill University. Hakan Erdogmus from the National Research Council of Canada has also collaborated on linking structural reuse with DASE.

12. References


