To appear:
Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Detroit, Michigan, May, 1995.

MANAGING COMPLEXITY IN HETEROGENEOUS SYSTEM SPECIFICATION, SIMULATION, AND SYNTHESIS

Asawaree Kalavade, José Luis Pino, and Edward A. Lee[1]

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720-1770.
{kalavade,pino,eal}@EECS.Berkeley.EDU

ABSTRACT

System-level design is characterized by a behavioral specification and heterogeneous hardware/software implementations. Exploring the design space is essential for good design. Specifying and managing complex design flows, tracking dependencies and tool invocations, and maintaining consistency of design data and flows are key issues that enable efficient design space exploration. In order to manage the complexity of this design process, an infrastructure that manages these issues, transparent to the user, is presented. These concepts have been implemented in the Ptolemy environment within a framework called DesignMaker. An example design flow for multiprocessor synthesis is presented in some detail to illustrate the features of DesignMaker. The end objective of the framework is to facilitate a flexible system-level codesign assistant.

1 INTRODUCTION

Typical applications of embedded systems include telecommunications, consumer products, robotics, and automotive control systems. Such embedded systems exhibit heterogeneity in implementation (hardware/software) as well as computational semantics (control/dataflow). As a result of increasing functional complexity, these systems are usually designed in a top-down manner, starting with a behavioral specification. The design of such heterogeneous hardware/software systems is often referred to as hardware/software codesign or system-level design. In this context, the key design problems are hardware/software partitioning, cosynthesis, and cosimulation [1].

A typical design flow for the hardware/software codesign problem [2] is shown in Figure 1. A behavioral-level design specification (ex: *modem.sdf*) is transformed into the final implementation, consisting of custom and commodity programmable hardware components and the software running on the programmable components, by passing through a sequence of tools. This is not a black-box push-button design process, but involves considerable user interaction. The user experiments with different design choices; design space exploration is the key to system level design. Managing the complexity of this design process is non-trivial. The features needed for efficient design space exploration include:

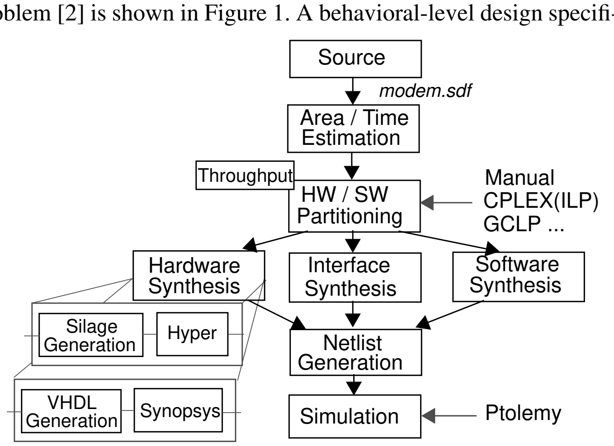

Figure 1. A typical design flow for the automated hardware/software codesign problem.

- *Modular and configurable flow specification mechanisms*

For example, in Figure 1, the user might be interested in first determining if a feasible partition exists. At this point only the *Estimation* and *Partition* tools need to be invoked; subsequent tools need not be run. Inefficiencies due to unnecessary tool invocations can be avoided if flows are specified modularly as in Figure 1.
A number of design options are available at each step in the design process. For instance, the *Partition* tool can be one of: a human-intervened manual partitioning, an exact but time consuming tool such as CPLEX using integer linear programming techniques, or an efficient custom optimized algorithm such as GCLP [3]. Depending on the available design time and desired accuracy, one of these is selected. This selection can be done either by the user, or by embedding this design choice within the flow. A design flow with a configurable methodology is thus easily extensible.

- *Mechanisms to systematically track tool dependencies and automatically determine the sequence of tool invocations*

After developing a particular design, the user might want to experiment with other options, for instance, different hardware synthesis mechanisms. One possible approach to hardware synthesis is Silage code generation followed by Hyper [4]. An alternative path is VHDL code generation followed by Synopsys tools [5]. If a specific tool is changed on the fly, the entire system need not be re-run; only those tools that are affected should be run (in this case: *Hardware Synthesis, Netlist Generation, Simulation).*

- *Managing consistency of design data, tools, and flows*

Detecting incompatibilities between tools and maintaining versions of the tools and design flows is necessary.

In this paper we propose an infrastructure that manages these aspects of the system-level design methodology. The end goal is to use this infrastructure to build a codesign system. Section 2 briefly mentions mechanisms for specification, simulation, and synthesis of heterogeneous designs. Section 3 presents the underlying concepts of design methodology management. Section 4 discusses implementation details and gives an example design flow that demonstrates the viability of our approach.

1. This research is part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program F33615-93-C-1317), SRC (94-DC-008), NSF (MIP-9201605), Office of Naval Technology (via Naval Research Laboratories), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, Rockwell, Sony, and Synopsys.

2 SPECIFICATION, SIMULATION, AND SYNTHESIS

Ptolemy [6] is an environment for simulation and prototyping of heterogeneous systems. Instead of trying to capture all possible models of computation into one all-encompassing model, the Ptolemy kernel implements an open architecture that enables an unlimited number of extensible models to be defined. Each model (called a "domain") is responsible for implementing its own data processing and data exchange strategies.

Heterogeneous systems can be specified using different levels of abstraction and/or semantics for the various subcomponents. For instance, a multimedia telecommunication system can be specified as a combination of an event-driven subsystem representing the packet-switched network and a dataflow subsystem modeling the signal processing components. Ptolemy supports multi-paradigm simulation in which different computational models co-exist; e.g. a dataflow system can interact with a finite-state machine component, or a hardware system can interact with software. Finally, heterogeneous systems can be synthesized using the hardware and software synthesis mechanisms reported in [2][4][7]. We have developed a mechanism to combine diverse schedulers (with different optimization objectives) for the software synthesis process [8]. This simplifies the software scheduling problem and also enables the use of specialized schedulers. This paper will focus on managing the complexity in these phases of the design process.

3 DESIGN METHODOLOGY MANAGEMENT

A design methodology specifies a sequence (flow) of tools that operate on data. Design methodology management (DMM) is formally defined as "definition, execution, and control of design methodologies in a flexible and configurable way" [9]. The problems encountered in DMM are: data, tool, and flow management.

DMM as such is not new; traditional DMM systems (often referred to as "frameworks") are used quite extensively in the physical VLSI design process. These systems [10][11] focus on data and tool management, i.e., maintaining consistent versions of data, and invoking a user-specified tool after ensuring that the preconditions for enabling it are satisfied, respectively. The key to system-level design, however, is design space exploration. At the system level, design flows are less well-defined than at the physical design level, and the range of tools involved is much larger. Powerful constructs for flow definition, dependency analysis, and automated flow execution dominate the system-level design process. *Our focus is to manage the complexity of the "system-level" design problem, with emphasis on flow management*.

3.1 Flow, Tool, and Data Model

Figure 2 illustrates the details of the components of our DMM mechanism. A design *flow* is specified as a directed graph, where nodes represent tools, and arcs specify the ordering between tools. *Tools* communicate via filenames and encapsulate actual programs. Tool parameters specify the arguments for these programs. A tool's inputs and outputs are associated with its *ports*. Ports can be of two types: *required* and *optional*. Optional ports make it possible to represent conditionals and iterations in flows (illustrated in Section 4.2).

The information model is represented as a distributed data structure. Associated with each tool is a `Param_Changed_Flag`

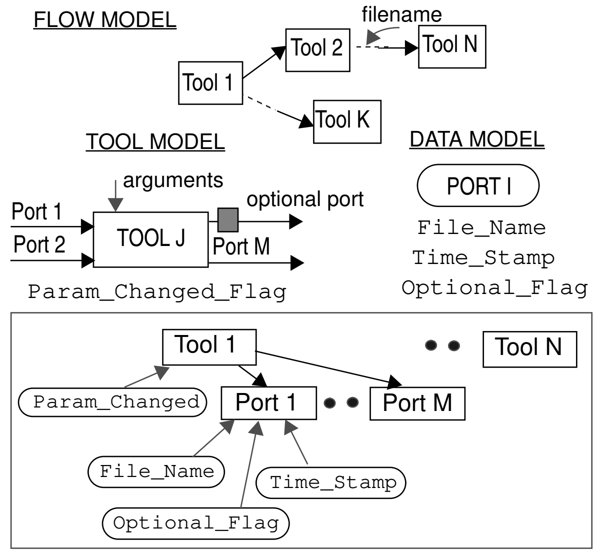

Figure 2. Components of a DMM system.

that gets set when parameters of a tool are changed. Associated with each port of a tool are three attributes: `File_Name`, `Time_Stamp`, and `Optional_Flag`. `File_Name` and `Time_Stamp` represent the filename and the timestamp of the data on the port (associated with the most recent invocation of the tool). `Optional_Flag` indicates whether the port is required or optional.

3.2 Dependencies

Figure 3 shows the three types of dependencies that are supported. *Temporal* dependencies track the timestamps on input-output ports of the tools — a tool needs to be run if any of its output is out-of-date, i.e., any of its input timestamps are newer than its output timestamps. *Data* dependency ensures that a tool is run whenever the file received on any of its input ports has either a filename or a timestamp that is different from the previous tool invocation. *Control* dependency tracks parameter changes; a tool needs to be run if any of its parameters has changed.

3.3 Flow Management

Automatic flow invocation is based on analyzing the tool dependencies and executing tools as required. A tool is said to be *enabled* when all of its required input ports have data. Absence of data on the optional input ports does not affect enabling. Once enabled, a tool is checked for dependencies. A tool is *invoked* (*run*) when at least one of its dependencies is live. On execution, a tool

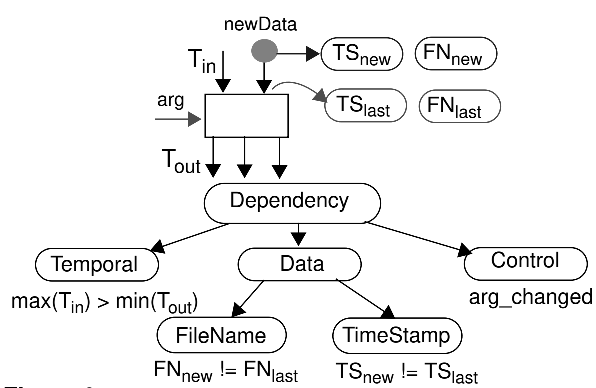

Figure 3. Dependencies used for tool management.

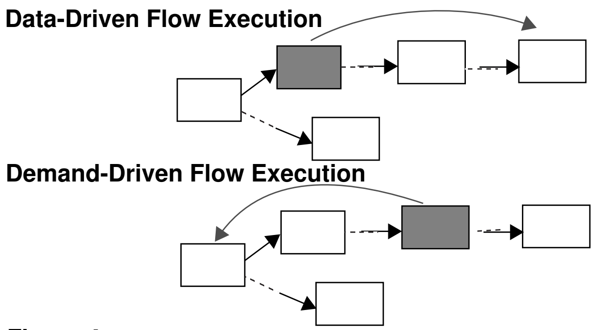

Figure 4. Flow execution mechanisms.

generates data on its required output ports, and conditionally on the optional output ports. Two types of flow invocation mechanisms are desired: data-driven, and demand-driven (Figure 4). In the data-driven approach, the flow scheduler traverses the flow according to precedences. The process halts when all tools with live dependencies have been exhausted. In the demand-driven mode, the user selects a tool for execution. The scheduler traverses the predecessors and executes all tools with live dependencies on the path.

4 DMM: IMPLEMENTATION

4.1 Implementation details

The DMM mechanism described in Section 3 has been implemented in the Ptolemy environment as a separate *domain*. Design flows are specified as graphical netlists. Conditionals and iterations are supported in the flow definition. Flows can be described hierarchically. Tools are encapsulated within basic blocks. Tools can have required and optional ports as well as parameters. Tool encapsulation involves writing scripts that call various programs. Tools can invoke programs on remote filesystems as well as programs with their own GUIs. The tool writer need not worry about the underlying timestamps and filenames. The DMM attributes and flow netlists are stored within the Oct database [12]. Tools communicate via files. A type resolution mechanism checks for data type compatibility between tools. The flow and tool manager is called *DesignMaker. DesignMaker* supports both data-driven and demand-driven flow execution. It resolves tool dependencies and automatically invokes tools. The underlying scheduler is a combination of dynamic dataflow and event-driven schedulers. *DesignMaker* detects deadlocks when iterative flows are incorrectly specified (such as forward dependencies on required inputs). *DesignMaker* also supports other features such as annotation of the design process and the display of the flow status (blocked, running, and ready) at any point in the design process.

Our motivation for implementing the DMM system in Ptolemy is to take advantage of the existing modular kernel and data structures, in which DMM fits seamlessly as a domain.

4.2 Example

Figure 5 illustrates a particular design flow example within DesignMaker. It represents the design flow for the automated synthesis of the hardware (netlist) and the software (programs running on the processors) components of a multiprocessor system that implements a given application. A dataflow graph *G* (*GraphName*) describing the application (ex: music synthesis) and a specific interprocessor communication mechanism (*Architecture*) are specified by the user. The goal is to synthesize this multiprocessor system with the minimum number of processors such that required *latency* is met. We now run through a typical flow execution sequence to demonstrate the effects of the various constructs and dependencies.

Suppose that the flow is run the very first time using *Run All*. Tools are examined for active dependencies by traversing the flow according to precedence ordering between tools. *Source (T1)* outputs the dataflow graph *G* specified by *GraphName*. *NumProcEstimator* (*T2*) and *Code Generator* (*T6*) are dependent on *T1*. Note that *T2* is enabled, while *T6* is not (second input *N*, has not yet been generated). *T2* is a hierarchical description of the estimation process, which iteratively determines the minimum number of processors required to implement *G* at the desired *latency*.

ProcEstimator (T3) estimates the number of processors (*N*) required to implement *G*. Estimators of different accuracy can be selected by changing parameters of this block. Suppose that the desired latency is 320 cycles and the sum of execution times of all the nodes in *G* is 900. A possible estimator would assume maximum parallelism to estimate a lower bound of 3 processors. *T3* has an *optional* input that receives an indication as to whether or not the current *N* satisfies the latency requirements. When this input is available, *T3* uses it to improve the estimate for *N*. Note that the loop (*T3-T4-T5*) does not deadlock because this input is optional. The *Scheduler* (*T4*) schedules *G* onto *N* (=3) processors and determines the actual time required (*makespan M*) to implement *G*. *T4* detects convergence of the estimation loop if the value of *N* generated by *T3* in consecutive iterations is the same. *T4* generates outputs conditionally — it generates output *N* if the iteration has converged, else it generates output *M* and the iteration *T5-T3-T4* is repeated. Different scheduling algorithms can be selected by changing parameters of *T4*. Suppose it computes the makespan to be 350. The *Comparator (T5)* compares the latency (320) and *M (350)* to generate the control signal for *T3*. *T3* is enabled by input received on its optional input, and refines its estimate of *N* to 4. Note that up to this point, as the system is being run the very first time, filename type data dependencies are active for all the blocks. (Data dependencies identify a change in the input data to a tool with respect to its previous invocation.) For *T4* however, filename data dependency is no longer active (since the same filename is retained), but temporal dependency becomes active. Recall that temporal dependencies detect out-of-date output data. As the input to *T4* is newer than its output, it runs again, recomputing *M* (say, 290). The sequence *T5-T3-T4* repeats. The estimation loop converges as the latency is now met.

The *Code Generator* (*T6*) is now invoked. *T6* uses the *N-processor* schedule to synthesize software for the *N* processors. A parameter for *T6* is used to select the type of code generator. *T7* generates the netlist for the system. The generated architectural model, where the processors run the synthesized software is then simulated by the *Simulator* (*T8*). As no more dependencies are alive, the flow execution stops. Subsequent changes to the flow or data could render parts of the flow invalid. For instance, suppose that the input to *T8* is modified externally and the *Run All* command is re-issued. The timestamp on the input to *T8* is different from that in its most recent invocation, causing the timestamp data dependency to be activated. As no other dependencies are active, only *T8* is invoked. If the system is run again and no data files or arguments have changed, then no tool is invoked.

Conditionals prevent unnecessary tool invocations. For instance, although *T6* and *T7* are data-dependent on *T4*, they are

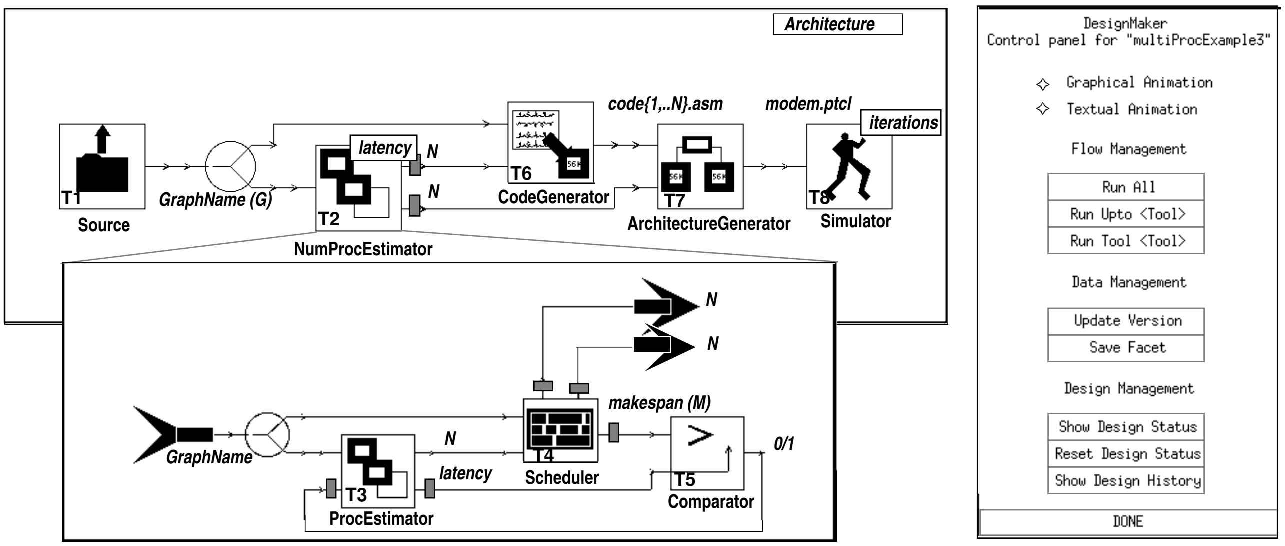

Figure 5. An example design flow for multiprocessor synthesis, specified within *DesignMaker*. The goal is to synthesize the hardware and software, with minimum number of processors, for a specified latency. The control panel shows some supported features.

invoked only when there is new data on their inputs and not unconditionally whenever *T4* runs. Control dependencies track parameter changes. For instance, if *GraphName* is changed, *T1* is first invoked due to a control dependency. This activates a filename data dependency for T2 as the input data (*GraphName)* changes, causing it to be invoked. Other tools on the downstream flow that are data-dependent on *T2* (complete flow) are similarly invoked. Suppose that the type of code generator is changed by changing a parameter for *T6*, only *T6* and its downstream tools (*T7, T8*) are invoked. The *Run Upto* option implements the demand-driven flow scheduler, while *Run Tool* allows a specific tool to be invoked by the user (it is run only if enabled).

Note than the DesignMaker is much more powerful than a graphical *make* [13] utility. Specification of iteration, hierarchy, and conditionals in the design flow, allowing optional inputs and outputs for tools, ensuring tool compatibility, and detecting argument changes are some of the additional features. The DesignMaker environment also makes it possible to track designs and design flows, store a library of flows, and maintain a history of the design.

5 CONCLUSIONS

System-level design deals with a behavioral specification and heterogeneous hardware/software implementations. Due to the wide range of design possibilities, efficient design space exploration is important. Specifying and managing complex design flows, tracking dependencies and tool invocations, and maintaining consistency of design data/flows are critical issues in this context. To manage the complexity of this design process, we have presented an infrastructure for design methodology management (DesignMaker) that manages these issues, transparent to the user.

We intend to apply this framework to develop a flexible hardware/software codesign system, as shown in Figure 1. It is also possible to embed intelligence in the exploration process using this infrastructure. We are looking at extensions where estimates are used by the system to automatically configure design flows.

6 ACKNOWLEDGEMENTS

The authors would like to thank Pratyush Moghe and Brian L. Evans for their helpful comments on this paper.

7 REFERENCES

[1] A. Kalavade, E. A. Lee, "Manifestations of Heterogeneity in Hardware/Software Codesign", *Proc. of the 31st Design Automation Conf.*, San Diego, CA, June 1994, pp. 437-438.

[2] A. Kalavade, E. A. Lee, "A Hardware/Software Codesign Methodology for DSP Applications", *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 16-28, Sept. 1993.

[3] A Kalavade, E. A. Lee, "A Global Criticality/ Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem", *Proc. of CODES/CASHE, Third Intl. Workshop on Hardware/Software Codesign*, Grenoble, France, Sept. 22-24, 1994, pp 42-48.

[4] J. M. Rabaey et al. "Fast Prototyping of datapath-intensive Architectures", *IEEE Design and Test of Computers*, pp. 40-51, June 1991.

[5] Synopsys Tools, Synopsys Inc., 700 East Middlefield Rd., Mountain View, CA 94043.

[6] J. Buck, S. Ha, E.A. Lee, D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *Int. Journal of Comp. Simulation*, special issue on "Simulation Software Development", v4, 155-182, Apr. 1994.

[7] J. Pino, S. Ha, E. Lee, J. Buck, "Software Synthesis for DSP Using Ptolemy", invited paper in *Journal on VLSI Signal Processing*, special issue, "Synthesis for DSP", to appear: 1994.

[8] J. L. Pino, E. A. Lee, "Hierarchical Static Scheduling of Dataflow Graphs", to appear: *Proc. of ICASSP, 1995.*

[9] S. Kleinfelft, M. Guiney, J. K. Miller, and M. Barnes, "Design Methodology Management", *Proc. of the IEEE,* vol. 82, no. 2, pp. 231-250, Feb. 1994.

[10] W. Allen, D. Rosenthal, K. Fidule, "The MCC CAD Framework Methodology Management System", *Proc. of the 28th Design Automation Conference,* 1991, pp 694-698.

[11] K. O. ten Bosch, P. Bingley, P. van der Wolf, "Design Flow Management in the Nelsis CAD Framework", *Proc. of the 28th Design Automation Conference*, June 1991, pp 711-716.

[12] D. Harrison, P. Moore, R. Spickelmier, A. R. Newton, "Data Management and Graphics Editing in the Berkeley Design Environment", *Proceedings of the International Conference on Computer Aided Design (ICCAD),* 1986, pp 24-27.

[13] S. Feldman, "Make — A Program for Maintaining Computer Programs", *Software Practice and Experience,* 1979, Vol. 9, pp 255-265.