5

DESIGN SPACE EXPLORATION

In the previous chapters we have discussed specific solutions for automated partitioning and synthesis. There we assumed a particular design objective and target architecture. In general, the system-level design problem cannot be posed as a single well-defined optimization problem from the designer's perspective. Typically, the designer needs to explore the possible options, tools, and architectures, choosing either automated tools or manually selecting his/her choices. This design space is quite large. As shown in Figure 5.1, a large number of target architectures and implementation technologies could be used to implement a system. Several optimization objectives and constraints are possible. The user also has access to a large number of design tools. The user might experiment with the design parameters, the target architectures, the optimization criteria, the tools used, or the sequence in which these tools are applied. As such, there is no *hardwired* design methodology. System-level design requires an *infrastructure* that *supports* efficient design space exploration — such an infrastructure permits considerable flexibility and at the same time relieves the user of the book-keeping.

As discussed in Chapter 1, tools that aid in design space exploration fall into two categories: estimation and management. Estimation tools are primarily used for *what-if* analysis, i.e., they give quick predictions on the outcome of applying certain synthesis or transformation tools. Management tools are required to orchestrate the design process, i.e., for systematic control of the design data, tools, and flow. In this chapter we focus on the management aspects of design space exploration; this is often referred to as *design methodology management*[1].

In Section 5.1, some of the related work in this area is discussed. In Section 5.2, we identify some of the requirements of a design methodology management framework. An infrastructure that supports these requirements is proposed in Section 5.3. We have implemented this infrastructure within the Ptolemy environment.

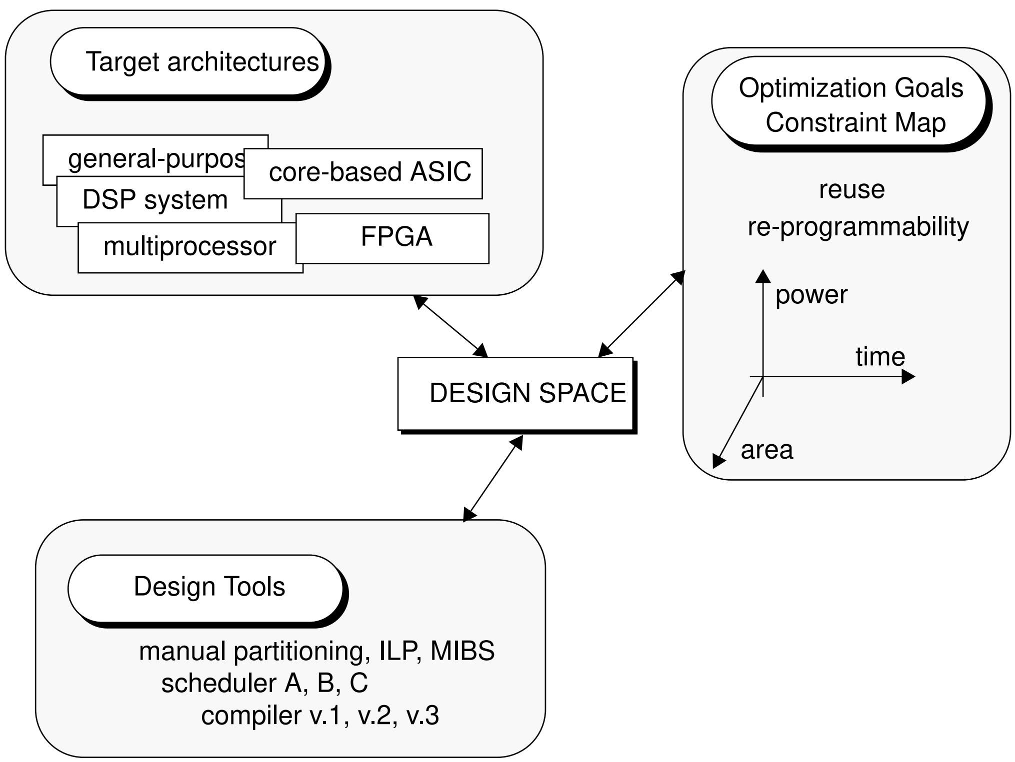


Figure 5.1. The design space.

1. Design methodology is defined as "the processes, techniques, or approaches employed in the solution of a problem". Design methodology management (DMM) is formally defined as "definition, execution, and control of design methodologies in a flexible and configurable way" [Kleinfelft94].

Some details of the implementation are presented in Section 5.4. In Section 5.5, we discuss the operation of the design methodology management framework with the help of two representative design flows.

5.1 Related Work

Design methodology 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. Kleinfelft *et al.* [Kleinfelft94] provide an exhaustive summary of the various research activities in this area. We mention only a few representative systems here.

The three main problems in design methodology management involve data management, tool management, and flow management.

The DMM systems used in the physical VLSI design process focus primarily on data management (i.e., maintaining consistent versions of data) [Harrison86] and tool management (i.e., invoking a user-specified tool after ensuring that the preconditions for enabling it are satisfied) [Chiuch90][Bosch91] [Brockman91]. Commercial CAD frameworks such as the Falcon framework [MentorGraphics2] also assist in tool and data management. The NELSIS framework [Bosch91][VanDerWolf93] provides a systematic representation and management mechanism for data and tools within a semantic database. CFI [CFI] defines standards for tool encapsulation and data models. As of this writing, no standard for flow management has been proposed by CFI.

The MMS framework [Allen91] focusses on distributed tool execution and multi-user environments. Recent efforts address the flow management problem. Some efforts approach the flow management problem from an AI angle [Knapp86]

[Bushness89], where the methodology and firing rules are stored in a knowledge-base and an inference engine determines the tool execution sequence. Yoda [Dewey89], a filter design system, has a knowledge-base of predictors (estimators). Predictors are used to determine the outcome of applying a particular tool and the results from such an exploration are used to construct a design plan (similar to a script), with feedback from the designer. The generated design plan is then automatically executed, where the actual tools are run. A trace-driven approach is proposed in [Casotto90], where a sample design session (the sequence of tools run by the user) is saved and future design sessions can be automatically controlled by following this trace.

CODES [Buchenrieder92] is a framework for codesign. It provides an open architecture for the integration of commercial and proprietary tools. A graphical representation of a design flow is translated to an internal Petri net representation. This is analyzed to determine firing rules. A codesign manager invokes the tools based on these firing rules.

Bentz et al. [Bentz95] present an information-based approach to design. They combine some of the traditional design methodology management features (such as data and tool management) with novel features for design space exploration via a tool called *Design Agent*. The *Design Agent* assists the user in collecting and managing information about a design. The designer queries for specific feedback (example: "what is the area of this FIR filter when implemented as an ASIC in a particular technology?", or "what is the code size when implemented on a particular DSP?"), rather than explicitly performing tasks to obtain this information. The *Design Agent* computes the information based on *actions* (sequence of tools) specified its database. These actions are context-specific, that is, different sets of actions apply to different design *domains* (such as ASIC design and DSP design).

5.2 Design Methodology Management: Requirements

Our focus is primarily on design flow specification and management. We do not address issues of database management, multi-user operation, and distributed tool execution. Our goal is to develop a framework that simplifies the designer's tasks by (1) providing mechanisms that allow the design flow to be specified in an intuitive way, (2) automatically invoking the tools in the design flow whenever possible, and (3) managing the infrastructure when the user decides the sequence of tool execution. The key features required for this are discussed next.

Flow Specification

Since tools involved in the system-level design process are often computationally intensive, it is important to avoid unnecessary invocation of tools. This requires that the design flow be specified in a *modular* way so that only the desired tools may be invoked. It should also be possible to specify the design flow hierarchically, in order to retain design modularity.

Specification of the flow also requires *iterative* and *conditional* constructs. Consider an example where an application is to be mapped to a multiprocessor system. Assume that the number of processors is not known apriori. The design sequence usually is to (1) estimate the number of processors required, (2) schedule the application onto these processors and compute the resultant throughput, (3) if the throughput does not meet the desired throughput, repeat (1), else continue with the remaining parts of the design such as code generation and netlist generation. To express such a design flow, the flow specification mechanism should support constructs such as conditionals and iterations.

A number of design tools are available for each step in the design process.

Consider hardware-software partitioning for example. If the application involves few components that have obvious mappings, partitioning could be done manually. If the mappings are not obvious, an exact and time-consuming integer-linear programming approach could be used to determine the optimal mappings. On the other hand, if the application is quite complex and there are several options available for implementing the different components, an efficient heuristic (such as the MIBS algorithm) could be used. If the design flow is *parameterizeable*, a new flow need not be developed for each type of tool used. Depending on the design size, the available design time, and the desired accuracy, one of the tools can be selected. This selection can be done either by the user, or by embedding this design choice within the flow.

Flow Execution

A designer should not have to keep track of the tools that have already run and those that need to be run. When the parameters or the data associated with a tool change, the entire design flow need not be re-run; only the affected tools should be run again. Keeping track of the tools that need to be run is quite cumbersome. A mechanism that automatically determines the sequence of tool invocations is needed. This calls for a mechanism much like a graphical "make" utility [Feldman79].

Flow Management

Different types of tools, with varying input and output formats, are used in the system-level design process. In the very least, a mechanism to automatically detect incompatibilities between tools is required. Data translators could also be invoked automatically.

Versions of tools and design flows also need to be maintained; it is not sufficient to just keep track of versions of data.

5.3 Infrastructure

We have developed an infrastructure that tries to support most of the requirements discussed in the previous section. Details of the infrastructure are discussed in this section. In Section 5.3.1, we present the underlying models used to specify the design flow, tools, and data. We use the term *dependency* to qualify the conditions that require a tool to be invoked for execution. In Section 5.3.2, we identify different types of dependencies. The flow execution mechanism analyzes the dependencies and automatically invokes tools within a design flow. Details of the flow execution mechanism are presented in Section 5.3.3.

5.3.1 Flow, Tool, and Data Model

Figure 5.2 illustrates the user's view of the design flow and tools. The design *flow* is specified as a directed graph, where nodes represent tools, and arcs specify the ordering between tools[2].

Tools encapsulate actual programs. Tool parameters specify the arguments for these programs. A tool can have multiple input and output ports. The ports are used to transfer filenames between tools. A "source" tool (such as a signal genera-

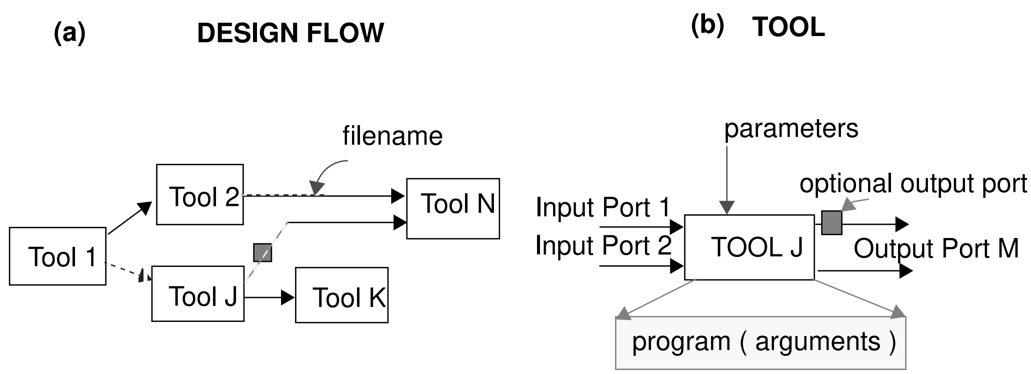


Figure 5.2. User's view of the design flow and tools.

2. Certain restrictions are imposed on the design flow in our implementation so as to avoid possible nondeterminacy. Refer to Section 5.4.

tor) has no input ports, while a "sink" tool (such as a display tool) has no output ports.

Ports can be either *required* or *optional*. The difference between required and optional input ports is that a tool cannot run unless it has data (valid filenames) on all its required input ports. A tool can run even if data is absent on an optional input port. When a tool is run, it generates data on all its required output ports, but not necessarily on optional output ports. Optional ports facilitate the use of conditionals and iterations in flows[3]. To understand this, consider the multiprocessor synthesis example mentioned earlier. The design problem is to synthesize a multiprocessor system that meets a desired throughout with a minimum number of processors. A possible design flow is shown in Figure 5.3. The *proc_estimator* determines the number of processors required and is described hierarchically as shown in the figure. The *estimator* tool estimates the number of processors needed. The estimated number of processors *num_procs_est* is given to a *scheduler* that computes the actual throughput for this number of processors. The *comparator* compares the actual throughput to the desired throughput and sends feedback to the *estimator*. The *estimator* has both a required input (port *x*), and an optional

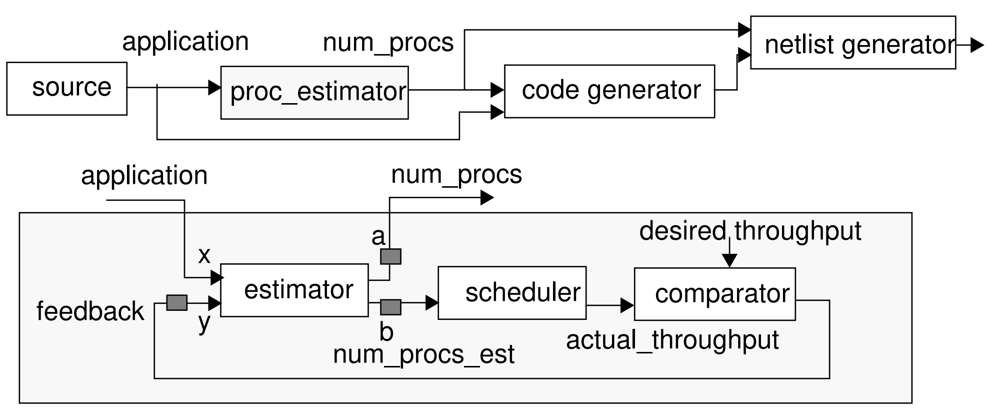


Figure 5.3. A design flow for the multiprocessor synthesis example.

3. Note that initial tokens can be used on arcs to equivalently model the functionality offered by optional ports.

input (port *y*). The *feedback* from the *comparator* is fed to the optional port of the *estimator*. If there is no feedback, the *estimator* estimates the number of processors based on its own information. If there is feedback, it updates its earlier estimate. Both output ports of the *estimator* are optional. When the estimation loop converges, the *estimator* generates data on the optional output port *a,* otherwise it generates the data for the revised estimate on the other optional output port *b*.

While optional ports permit modeling of a wide variety of applications, their use can potentially lead to nondeterminate behavior. In our implementation, we handle nondeterminacy by imposing strict restrictions on the allowed design flows for fully automated design flow execution. The flow scheduler identifies a potential nondeterminacy in a design flow and alerts the user. In most cases, the user knows what he/she had in mind when designing the design flow and can guide the system accordingly. The details of our implementation are described in Section 5.4.

Figure 5.4 shows the internal model of the tools and the data associated with a tool's input and output ports. Associated with each tool is a flag, called *Param_Changed_Flag,* which gets set when parameters of a tool are changed. Associated with each port of a tool are several attributes: $File\_Name_{last}$, $File\_Name_{new}$, $Time\_Stamp_{last}$, $Time\_Stamp_{new}$, and *Optional_Flag*. $File\_Name_{last}$ and $Time\_Stamp_{last}$ attributes store the filename and the timestamp[4] of the data on a port as of the earlier invocation of the tool. $File\_Name_{new}$ (and $Time\_Stamp_{new}$) represent the filename and the timestamp of the data updated on a port in the *current* invocation of the tool. *Optional_Flag* indicates whether the port is required or optional. The infrastructure stores the internal representation of the design flow as shown in Figure 5.4-c.

4. The timestamp indicates when the data associated with the file was last modified.

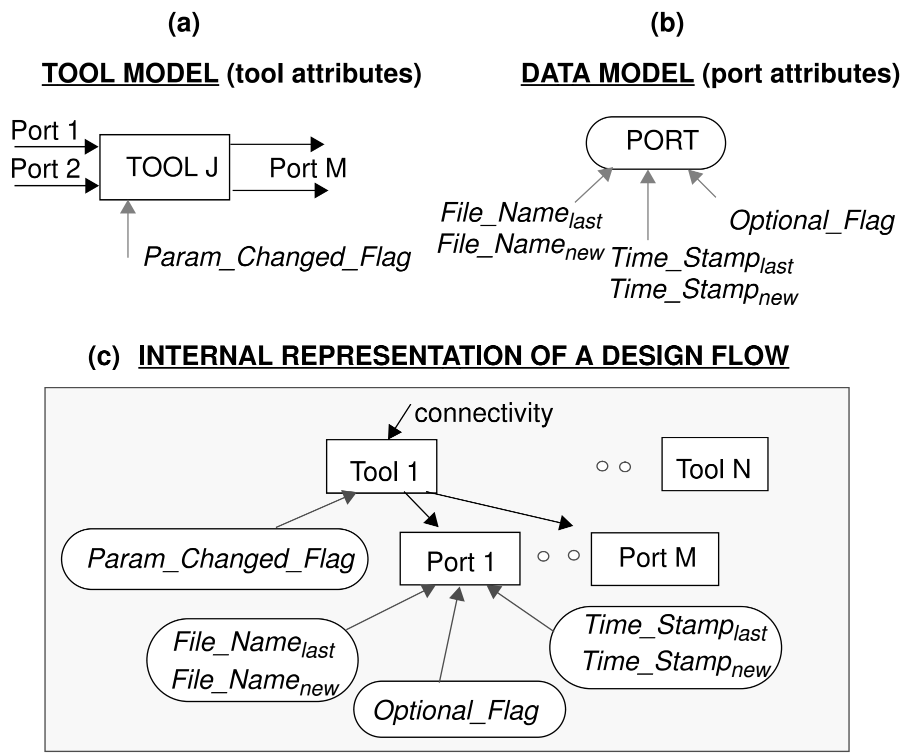


Figure 5.4. Internal representation of the tools and data.

5.3.2 Dependencies

The *Param_Changed_Flag*, and the filename and timestamp attributes are used to determine whether a tool needs to be run. For example, if the parameters of a tool have changed since its last invocation, the tool needs to be run. Similarly, if the filename associated with the current invocation of a tool is different from the filename associated with its previous invocation, this indicates a change in data, and requires the tool to be run again. We use the term *dependency* to qualify this behavior. Figure 5.5 shows three types of dependencies that are supported.

A *temporal* dependency tracks the timestamps associated with the data on the input ports of a tool. A tool needs to be run if its temporal dependency is alive, i.e., if the timestamp on any of its inputs is *newer* than the timestamp associated with the previous invocation of the tool, or if $Time\_Stamp_{new} > Time\_Stamp_{last}$. A

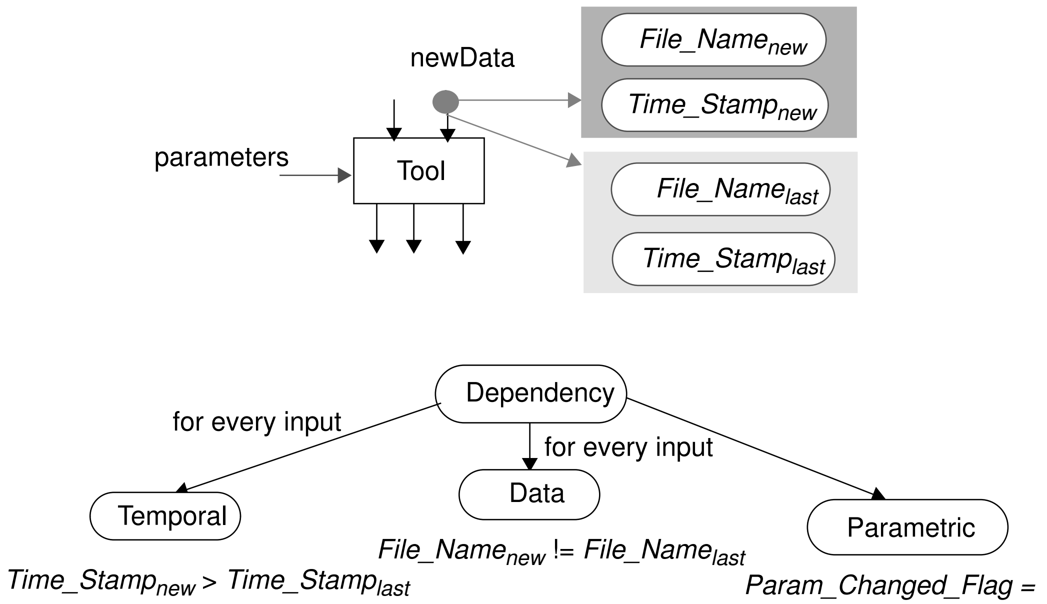


Figure 5.5. Dependencies used for tool management.

data dependency tracks changes in the filenames associated with the input ports of the tool. A tool needs to be run if its data dependency is alive, i.e., if the filename of the data associated with any input port is *different* from the filename associated with the previous invocation of the tool, or equivalently if $File\_Name_{last}$!= $File\_Name_{new}$. A *parametric* dependency tracks parameter changes; a tool is run if any parameter changes.

5.3.3 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 alive. On execution, a tool generates data on its required output ports, and possibly on its optional output ports.

As shown in Figure 5.6, two types of flow invocation mechanisms are desired: data-driven, and demand-driven. In the data-driven approach, the flow

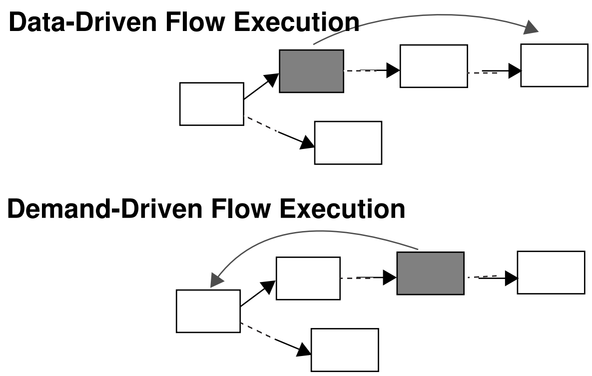


Figure 5.6. Flow execution mechanisms.

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. The details of the scheduler are given in the next section and in Appendix A9.

5.4 Implementation: DMM Domain within Ptolemy

We have implemented the design methodology management framework within the Ptolemy environment as a separate *domain* (called DMM domain). The details of the flow specification and execution mechanisms are described next.

Flow Specification

The design flow is specified as a graphical netlist (through the Ptolemy user interface, VEM). The netlist shows the connectivity between tools. In the current implementation, the flow is restricted to having a single "source" tool; multiple sources are not allowed. The flow can be described hierarchically. Figure 5.7 shows the design flow for the multiprocessor synthesis example described earlier. This flow will be discussed in more detail in Section 5.5.1. Optional ports can be used to define conditionals and iterations in the flow definition. However, this could lead to nondeterminate flows. A following section on the flow scheduler discusses the operation of the scheduler when this happens.

Tool Encapsulation

Tools are encapsulated within Stars (the atomic elements in Ptolemy, described in Section 4.2.1). Tool encapsulation involves writing scripts that call various programs. These programs could be either other Ptolemy functions or stand-alone executables. Figure 5.8 shows an example of tool encapsulation. It shows the *Code Generator* tool from the multiprocessor synthesis example. The *Code Generator* generates a multiprocessor implementation (multiple programs) for the input application. It uses a code generation routine within Ptolemy (*ptkGenCode*) to generate the code. The *Code Generator* has two inputs (both required): the application description (*graph*) and the number of processors (*numProcs*). The tool generates programs for all the processors. The output of the tool is specified as a file (*codeFileNames*) containing the names of the generated pro-

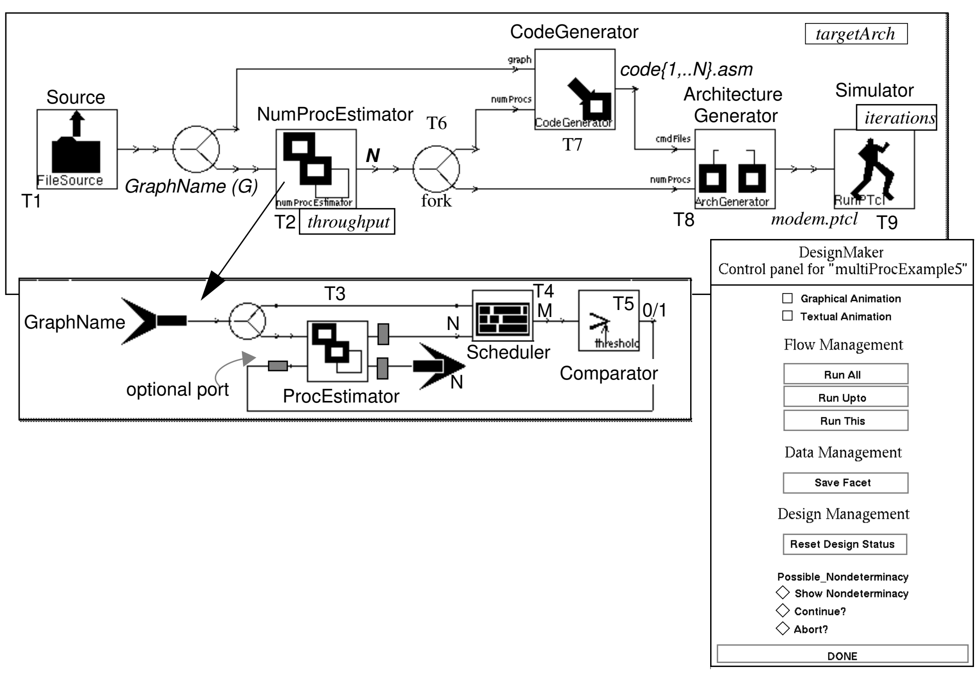
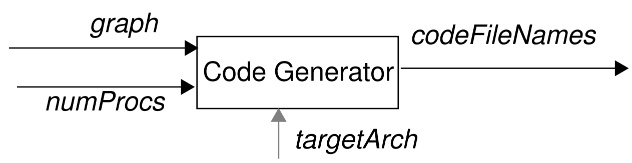


Figure 5.7. The design flow for the multiprocessor synthesis example, specified within the DMM domain in Ptolemy.

grams. The *Code Generator* has a parameter (*targetArch*) that specifies the assumed target architecture. The function "*go*()" contains the code that gets executed when the tool is invoked. When invoked, the tool first reads in the name of the file containing the graph (*graphName*). The functions *getName*(), *getDomain*() and *getHandle*() obtain the identifiers for the application pointed to by *graphName*. The number of processors is read into the variable *numberProcs* by scanning the input file *procFileName*. The Ptolemy routine that generates the code




```
defstar {
        name { CodeGenerator }
        domain { DMM }
        input { name { graph } }
        input { name { numProcs } }
        output { name { codeFileNames } }
        state { name { targetArch } default { sharedMemory } }
        go {
            // get application name and identifiers
              graphName = graph.getFileName();
              name = getName( graphName );
              domain = getDomain( graphName );
              handle = getHandle( graphName );
            // get number of processors
              procFileName = numProcs.getFileName();
              fp = fopen(procFileName, "r");
              fscanf(fp, "%d", &numberProcs);
            // run tcl command for code generation
              Tcl_VarEval(ptkInterp, "ptkGenCode", name, domain, handle,
                          targetArch, numberProcs);
            // generate output file names
              codeFileNames.putFileName(fout);
              fclose(fp);
        }
}
```

Figure 5.8. An example of Tool encapsulation: *CodeGenerator* tool.

(*ptkGenCode*) is then called with the application identifiers, the target architecture, and the number of processors. The result is written into the file *codeFileNames*.

Flow Scheduler (*DesignMaker*)

The tool writer need not worry about the underlying timestamps and filenames. The DMM attributes (*Param_Changed_Flag*, $Time\_Stamp_{last}$, $File\_Name_{last}$, $Time\_Stamp_{new}$, $File\_Name_{new}$, and *Optional_Flag*) and flow netlists are stored within the Oct database [Harrison86]. The flow manager called *DesignMaker* analyzes these attributes and automatically invokes the tools. Figure 5.9 shows the control panel associated with *DesignMaker*. There are three possible approaches to executing the flow: (1) The user opts to run the entire design flow (*Run All*), or (2) The user selects a certain tool upto which the flow should be executed (*Run Upto*), or (3) The user asks for a specific tool to be invoked (*Run This*).

In the *Run All* mode, the flow scheduler traverses the design flow, starting with the source tool, executing tools as necessary. To do this, it maintains a list of enabled tools. An enabled tool is checked for its dependencies and invoked for execution if any of its dependencies is alive. On execution, the appropriate descen-

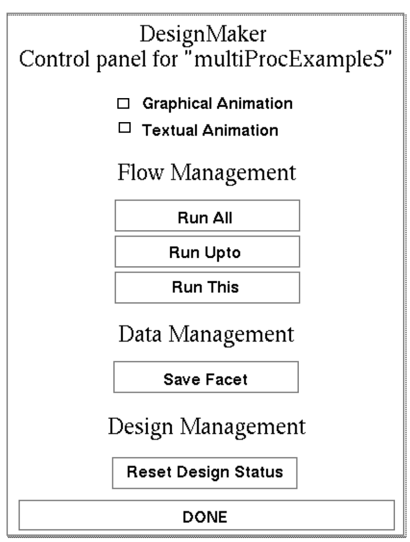


Figure 5.9. Control panel in the DMM domain.

dents of the tool are identified, and their corresponding input port is marked. If all the required input ports of a descendent tool are marked, the descendent tool is added to the list of enabled tools. Further details are presented in Appendix A9.2.

As mentioned earlier, it is possible that a flow may have a nondeterminate behavior[5] due to the presence of optional ports. Currently, the scheduler does very strict checking and conservatively flags flows that are possibly nondeterminate. The pseudocode for the algorithm for detecting nondeterminacy is given in Appendix A9.2. Some of these flagged flows can be determinate if the user has taken appropriate care in designing the flow. In such a case, the user can choose to use our flow scheduler at his/her risk. Alternatively, the user can schedule the tools manually in the sequence desired, by using the *Run This* mode.

In the *Run Upto* mode, the user selects a certain tool upto and including which the flow should be executed. All the predecessors of the selected tool are identified and tagged. The design flow is then traversed as in the *Run All* mode; only the tagged tools are executed. In the current implementation, we support this mode for acyclic flows only. The details of the scheduling algorithm are given in Appendix A9.3.

As mentioned earlier, the user can choose to manually schedule the flow by using the *Run This* mode. Details are given in Appendix A9.4.

5.5 Examples

The design methodology management domain and the operation of *DesignMaker* are described next with the help of two examples.

5. The generated data might not be independent of the sequence used to invoke the tools.

5.5.1 Multiprocessor System Design

Consider the design flow for the multiprocessor synthesis example, shown in Figure 5.7. A dataflow graph *G* (*GraphName*) describing the application (ex: music synthesis) and a specific interprocessor communication mechanism (*targetArch,* for example shared memory) are specified by the user. The goal is to synthesize this multiprocessor system with the minimum number of processors such that a required *throughput* is met. The synthesis process generates the hardware components (netlist) and the software components (programs running on the processors).

We now run through a typical flow execution sequence. Suppose that the required *makespan* (or 1/*throughput*) is 320 cycles.

When the *Run All* command is issued, the scheduler detects a possible nondeterminacy between tools *T4* and *T6* [6]and alerts the user accordingly. The control panel in Figure 5.7 shows the query for continuing or aborting the run. We know, however (by the way we have designed it), that the estimator tool *T3* (*ProcEstimator*) generates the output going to *T6* only after *T4* has finished. Hence, there is no nondeterminacy. We can then select the "continue" option to use the automated scheduling. The operation of the system under this selection is discussed next.

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* (*T7*) are dependent on *T1*. Note that *T2* is enabled, while *T7* is not (the second input *N* has not yet been generated). *T2* is a hierarchical description of the estimation process, which deter-

6. Since the nondeterminacy condition is (using the notation defined in Appendix A9) $T4 = P(T6)$, $T6 \neq P(T4)$, and $T4 \neq OP(T6)$.

mines the minimum number of processors required to implement *G* at the desired throughput by an iterative computation. The operation of the estimator is described next.

ProcEstimator (T3) estimates the number of processors (*N*) required to implement *G*. *T3* has an *optional* input that receives an indication as to whether or not the current *N* satisfies the throughput requirements. When this input is available, *T3* uses it to improve the estimate for *N,* and when it is not, *T3* computes the estimate from scratch. Note that the loop (*T3-T4-T5*) does not deadlock because this input is optional. At the start the feedback input is not available since *T5* has not run. In this case, *T3* computes the estimate based on its estimation algorithm. Estimators of different accuracy can be selected by changing parameters of this block. Consider a simple estimator that computes the estimate assuming maximum parallelism. Suppose that the sum of execution times of all the nodes in *G* is 900. *T3* estimates a lower bound of 3 (=900/320) processors. The *Scheduler* (*T4)* then schedules *G* onto *N* (*N*=3) processors and determines the actual time required (*makespan M*) to implement *G*. Different scheduling algorithms can be selected by changing parameters of *T4*. Suppose it computes the makespan to be 350. The *Comparator (T5)* compares the required makespan (320) and *M* (350) to generate the control signal for *T3*. *T3* is enabled by the 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, data dependencies are alive for all the blocks. For *T4* however, the data dependency is no longer active (since the same filename is retained), but temporal dependency becomes alive. Recall that temporal dependencies detect out-of-date data. As the timestamp on the input to *T4* is newer than the timestamp associated with the previous invocation of *T4*, T4 is re-run and it recomputes *M* (say, 290). *T5* sends a output of zero to *T3*. *T3* detects convergence of the estimation loop when the feedback from *T5* is zero, and sends the computed number of processors *N* to the fork *T6*. *T6* merely copies its input data to its two outputs.

Since its second input is available, the *Code Generator* (*T7*) is now invoked. *T7* synthesizes software for the *N* processors. *T8* generates the netlist for the system. The generated architectural model, where the processors run the synthesized software, is then simulated by the *Simulator* (*T9*). *T9* has a parameter (*iterations*) that indicates the number of cycles to simulate the design for. After *T9* runs, no live dependencies exist and the flow execution stops.

Subsequent changes to the flow or data could render parts of the flow invalid. For instance, suppose that the parameter "*iterations*" to *T9* is modified and the *Run All* command is issued. A parametric dependency is activated for *T9*. As no other dependencies are alive, only *T9* is invoked. If the *Run All* command is issued again, since no data files or parameters have changed, no tool is invoked. Next, suppose that *Run All* is issued after *GraphName* is changed. *T1* is first invoked due to a parametric dependency. This activates a 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* (the complete flow) are then invoked by the scheduler.

5.5.2 Design Assistant

The Design Assistant, described in Chapter 2, is a framework for system-level codesign. Figure 5.10 shows the Design Assistant implemented in the DMM domain.

Source (T1) outputs the SDF graph *G* specified by *GraphName* (say *simple.sdf*). *G* is then partitioned into hardware and software. Currently the Design Assistant supports manual partitioning.

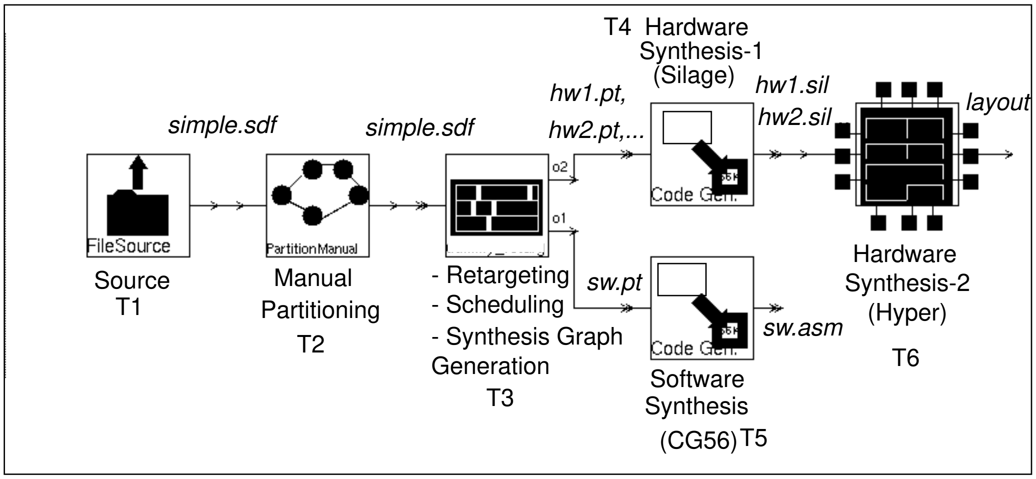


Figure 5.10. The Design Assistant implemented in the DMM domain.

Figure 5.11 shows the behavior of *T2* when configured for manual partitioning. The application, which is to be partitioned, is automatically displayed by the partitioning tool (Figure 5.11-a). The partitioning tool also brings up a selection panel to aid in manual partitioning (Figure 5.11-b). The user can then select parts of the application and assign them to hardware or software.

The next tool in the design flow is *T3*. *T3* first retargets the nodes in the SDF description to technology-dependent nodes (CG56 for software-mapped nodes, and Silage for hardware-mapped nodes). *T3* then inserts the *Send* and *Receive* nodes for communication across the hardware-software boundary. Since we have done manual partitioning, a global schedule needs to be generated. Given

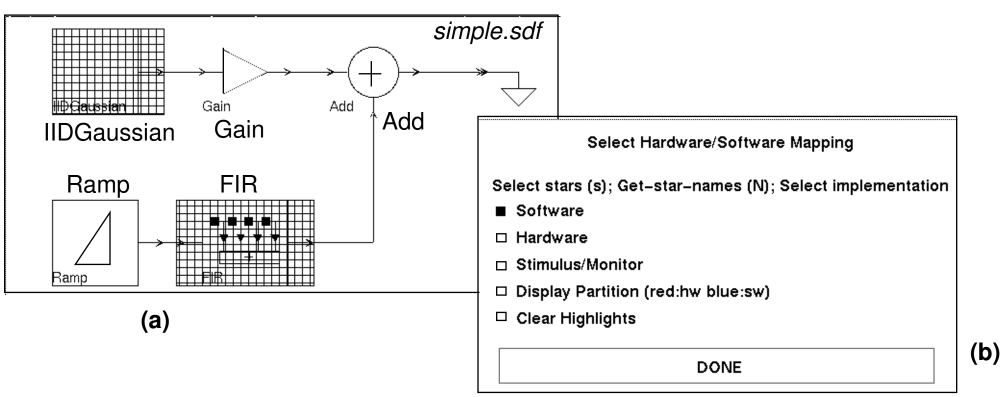


Figure 5.11. Run-time behavior of the Design Assistant with manual partitioning.

the assignment to hardware and software, *T3* generates a global schedule[7]. Finally, *T3* generates the hardware and software graphs using techniques described in Section 4.1.2. A separate hardware graph is generated for each node mapped to hardware. A single software graph is generated for all the nodes mapped to software. The software graph includes the *Send* and *Receive* nodes. *T3* uses the global schedule to derive a schedule for the nodes mapped to software. Figure 5.12 shows the outputs generated by *T3* for the graph *simple.sdf* shown in Figure 5.11. Suppose that the nodes *IIDGaussian* and *FIR* are mapped to software, and the nodes *Gain* and *Add* are mapped to hardware. The nodes *Ramp* and *Blackhole* correspond to the stimulus and monitor nodes which are not actually synthesized. Figure 5.12 (a) and (b) show the hardware graphs for the two hardware-mapped nodes.

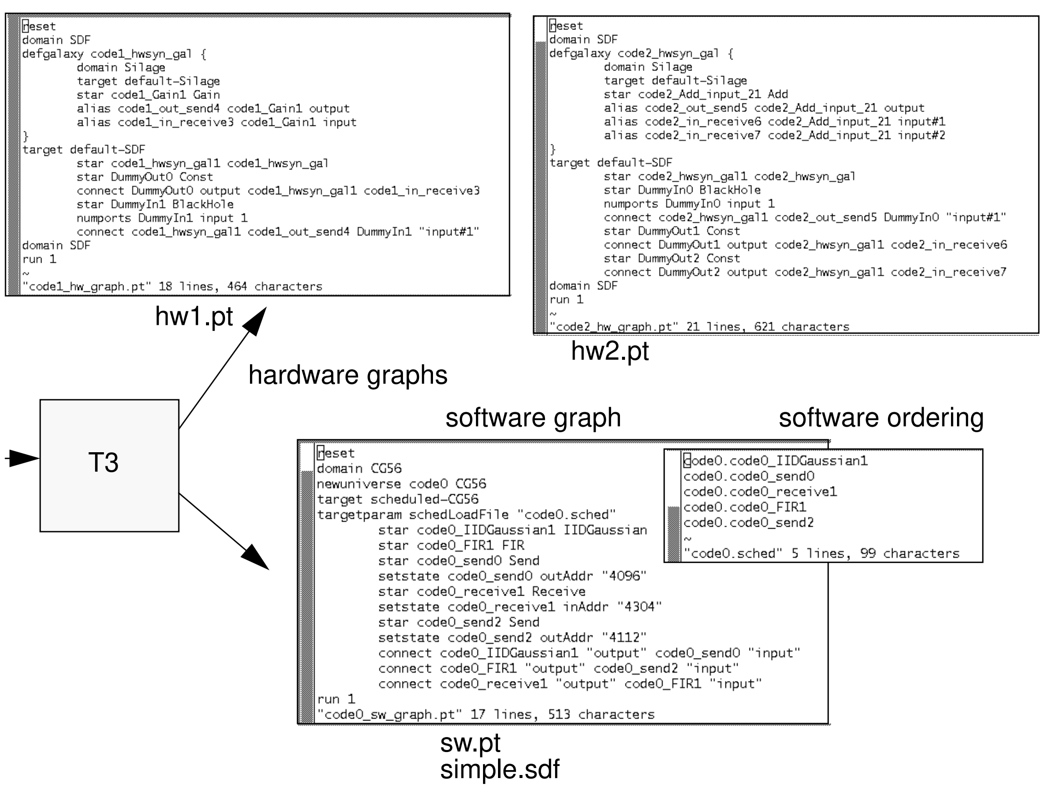


Figure 5.12. Hardware and software graphs generated by *T3*.

7. If the automated partitioning mechanism described in Chapter 3 were used, this step would not be needed since the automated partitioning algorithm generates the schedule.

Figure 5.12-(c) and (d) show the software graph and the software ordering respectively. Note the added *Send* and *Receive* nodes in the software graph and ordering.

T4 and *T5* are the hardware and software synthesis tools. *T4* generates Silage code for each hardware graph. *T5* generates a single assembly code file corresponding to all the software-mapped nodes. It uses the schedule generated by *T3* to order the nodes in the software graph. The outputs of *T4* and *T5* are shown in Figure 5.13.

T6 uses the generated Silage code to generated the final layout for the hardware components by running Hyper.

The current implementation of the Design Assistant has some restrictions:

1. It supports a predefined target architecture consisting of a single programmable processor and multiple hardware units. The current implementation assumes a Motorola 56000 target processor and Silage-based hardware

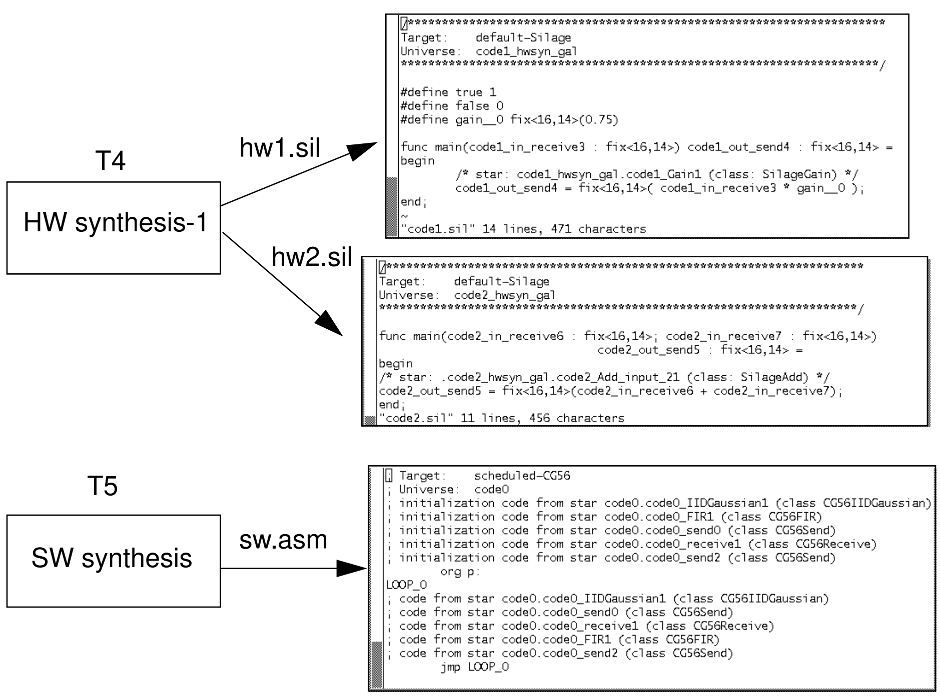


Figure 5.13. Silage and CG56 assembly code generated by the hardware and software synthesis tools.

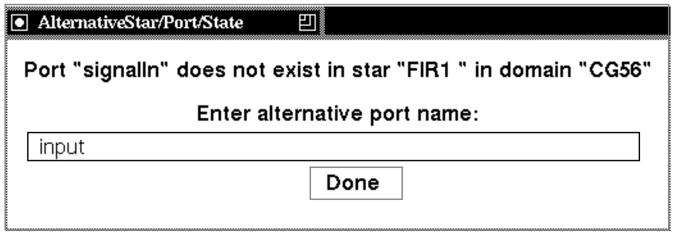


Figure 5.14. Support for retargeting.

synthesis, although it can be easily extended to support other processors or VHDL-based synthesis. Also, as mentioned in Chapter 4, we do not consider hardware reuse.

2. *T3* operates on a directed acyclic graph. It can be extended quite easily to support the full SDF model by simply adding a tool that converts a SDF graph to a directed acyclic graph.

3. The hardware-software interface is not automatically generated, although the techniques described in Section 4.1.5 can be automated quite easily.

4. The retargeting mechanism in *T3* assumes the existence of Silage and CG56 library elements for all the nodes in the SDF graph. We have implemented a rudimentary mechanism to query the user for alternative elements if a certain element does not exist in the corresponding implementation. The alternative element should, however, have the same number (names can be different) of inputs and outputs as the SDF (technology-independent) model. Figure 5.14 shows a simple scenario where the FIR node in *simple.sdf* is mapped to software. The names of the ports in the technology-independent model (SDF) and the technology-dependent model (CG56) are different (called *signalIn* and *signalOut* in the SDF domain and *input* and *output* in the CG56 domain). The system queries for an alternative port name to look for when doing the retargeting. A similar mechanism queries

for an alternative star if one doesn't exist for the desired implementation.

5.6 Summary

The system-level design space is quite large. As the number of tools and design possibilities increases, the design space explodes quite rapidly. Although a number of CAD systems for system-level design are now emerging [Chou94] [Kumar93][Theissinger94], most of them do not provide any support for managing the complexity of the design process; they contain point tools and leave the management aspects to the designer. We believe that managing the design process plays an equally important role in system-level design, as do the tools used for different aspects of the design. To this end, we have developed a framework that supports design methodology management. The design flow is considered a part of the design itself.

In this chapter, we have presented an infrastructure that supports efficient management of the design process, with emphasis on design flow management. This infrastructure contains powerful constructs for flow definition, dependency analysis, and automated flow execution. The infrastructure is developed as the DMM domain within Ptolemy. A tool called *DesignMaker* is responsible for automated flow management. Although *DesignMaker* derives its name from being a "*make*" utility for designs, it is much more powerful than a "graphical" *make* utility. Specification of iterations, hierarchy, and conditionals in the design flow, allowing optional inputs and outputs for tools, ensuring tool compatibility, and detecting parameter changes are some of the additional features.

We have integrated a number of point tools (such as tools for estimation, hardware-software partitioning, cosynthesis, and cosimulation) into this framework to form a complete codesign environment, which we call the Design Assistant.