3. In the partitioning and synthesis techniques, we assume that hardware is not reused between modules. The implications of this assumption are discussed in Chapter 4.

2.6 Summary

We propose the *Design Assistant* as a framework for system-level design. The Design Assistant is a unified platform consisting of tools for partitioning, simulating, and synthesizing mixed hardware-software systems. System-level design is not a black-box process. The designer may wish to experiment with different design parameters, tools, and flows. The Design Assistant supports efficient design space exploration; embedded in it is a design methodology manager that manages the design flow, tools, and data.

The next three chapters focus on the various aspects of the Design Assistant. In Chapter 3, an analytical framework for hardware-software partitioning is proposed. In Chapter 4, we discuss the approach used to cosynthesize and cosimulate a partitioned application. The design methodology management infrastructure is presented in Chapter 5.

components, and the software is the program running on the programmable component. The hardware-software interface, consisting of the glue logic and controllers, depends on the communication mechanism selected. The specifics of the interface are discussed in Chapter 4. This target architectural model subsumes both the design styles shown in Figure 2.4. Further details of this architecture are given in Section 4.1.1.

2.5 Restrictions in the Design Assistant

In summary, within the large design space, we restrict the applications of interest to be of the type shown in Figure 2.6. To reiterate the key limitations of our approach:

1. We restrict our techniques to the design of applications specified as SDF graphs. In particular, we assume that the DAG generated from such a SDF graph is the input to our partitioning and synthesis tools.

2. The target architecture is assumed to be of the type shown in Figure 2.5, it consists of a single programmable processor and multiple hardware modules.

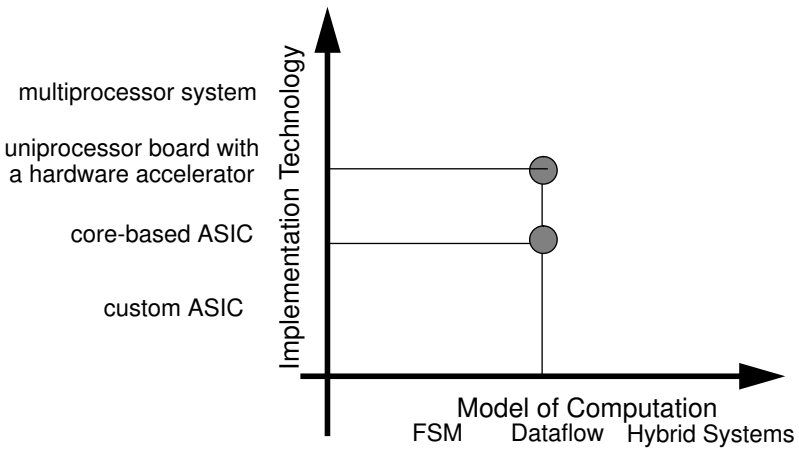

Figure 2.6. Specification semantics and target architecture.

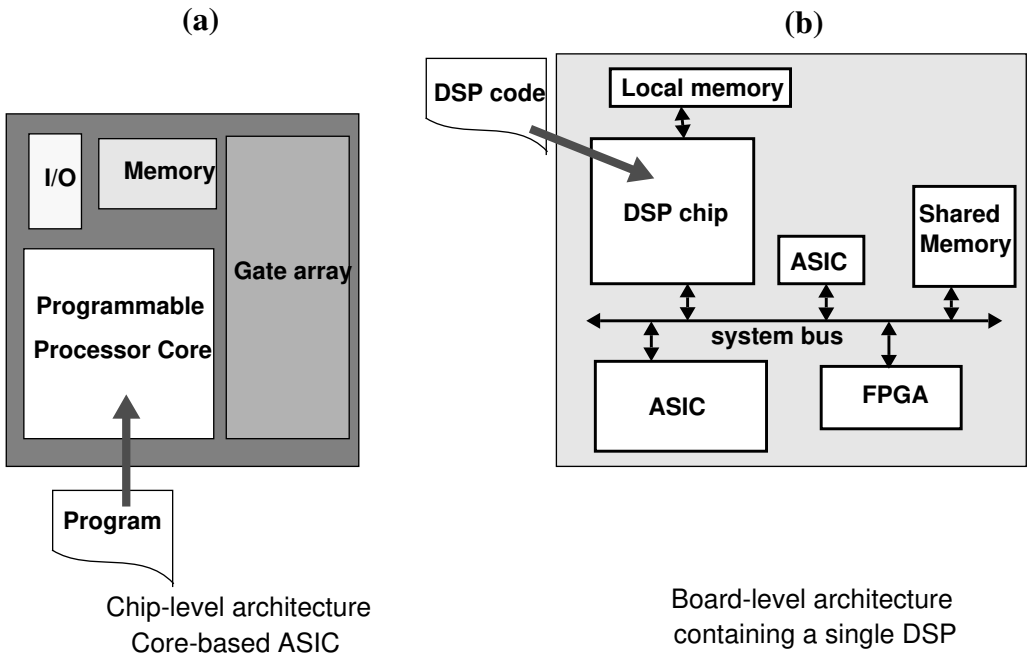

Figure 2.4. Typical target architectures for mixed hardware-software systems.

hardware and software components. These include FPGAs, ASICs, general purpose programmable processors, special purpose processors (DSP for instance), and domain-specific processors (processors optimized for a class of applications).

In this thesis we assume that the target architecture consists of a single programmable processor and multiple hardware modules. Figure 2.5 shows the chosen architecture. The hardware consists of the custom or semi-custom datapath

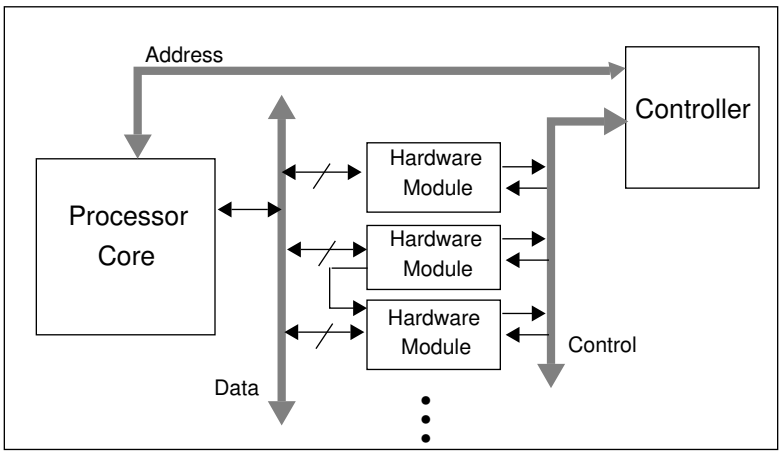

Figure 2.5. The target architecture.

that are represented in the SDF model of computation; the entire application itself will likely use a variety of models, however.

Figure 1.1 shows a typical task-level SDF specification in Ptolemy. The SDF graph can be translated into a directed acyclic graph (DAG), representing precedences between tasks, using techniques discussed in [Lee87]. This DAG is the input to the hardware-software partitioning algorithms. We assume that a hierarchically specified node in the SDF graph (such as the AGC in Figure 1.1) is not flattened when the SDF graph is converted to a DAG, i.e., it retains its hierarchical structure. The user can guide the partitioning process by choosing the hierarchy. The hierarchical representation of a node is called its *subgraph*.

2.4 Target Architecture

Although the Design Assistant does not assume a target architecture, the particular partitioning and synthesis tools used depend on the underlying target architecture. Figure 2.4 shows two typical architectures of mixed hardware-software systems. Figure 2.4-a illustrates a core-based ASIC. This is an emerging design style, where a programmable processor core is combined with a custom datapath within a single die [Bier95b]. Such core-based designs offer numerous advantages: performance improvement (due to critical components being implemented in custom datapaths, and faster internal communication between the hardware and software), field and mask programmability (due to the programmable core), and area and power reduction (due to integration of hardware and software within a single core). This architecture is especially attractive for portable applications, such as those typically found in digital cellular telephony. Figure 2.4-b illustrates a board-level architecture comprising several programmable and custom

four classes of semantics that can be used for abstract system-level modeling. A particular model may be better-suited for a certain part of an application, based on its properties. For instance, control-dominated components of an application are best described by hierarchical FSMs, while computation-dominated components are better-suited to dataflow semantics. FSM and dataflow semantics have been shown to be amenable to both hardware and software synthesis. Discrete-event models are commonly used for hardware modeling and are usually implemented in hardware. Another possible class of semantics is the imperative model, such as that found in C or C++. Such a model is usually better-suited for a software implementation. Thus, an application can be specified using a combination of these semantics, with each component subsystem being represented in a model of computation best suited for it.

We focus on periodic real-time signal processing applications with fixed throughput constraints. Many such applications can be specified in the synchronous dataflow (SDF) model of computation [Lee87] quite naturally. In this model, an application is represented as a graph, where nodes represent computations and arcs indicate the flow of data. A node fires when it receives data on all of its inputs, and on firing, it generates data on all of its outputs. SDF is a special case of dataflow, where the number of samples consumed and produced by a node firing is known statically (at compile time). This makes it possible to develop efficient compile-time scheduling techniques. The SDF specification supports manifest iterations, hierarchy, delays, feedback, and multirate operations. Synthesis of both hardware [Rabaey91] and software [Pino95a] from the SDF model of computation has been demonstrated and the model has proven useful for a reasonable set of signal processing applications.

In this thesis we focus on the design of the components of an application

invocations is needed. For instance, if the user changes the hardware synthesis mechanism (perhaps a standard-cell based design instead of one based on field programmable gate arrays), the system should only invoke the dependencies of the hardware synthesis tool (in this case: *Hardware Synthesis, Netlist Generation, Simulation*).

3. Managing consistency of design data, tools, and flows

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

   At the system level, it is no longer sufficient to keep track of versions of data — versions of tools and design flows also need to be maintained.

A *design methodology management* infrastructure that supports these requirements is described in Chapter 5.

2.3 Specification Semantics

A complete system-level design is likely to involve various subsystems, each of which is specified in a different model of computation. Figure 2.3 shows

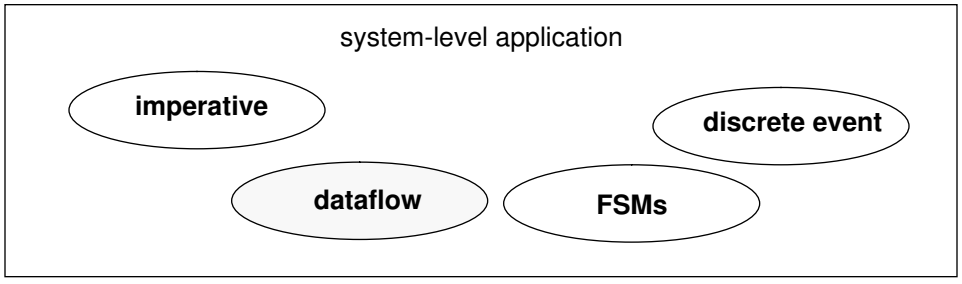

Figure 2.3. Models of computation used in system-level modeling.

2.2 Requirements for Design-Space Exploration

As discussed in Chapter 1, the design process is not a black-box push-button process; it 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:

1. Modular and configurable flow specification mechanisms

   In Figure 2.2, the user might be interested in first determining whether a feasible partition exists. At this point only the *Estimation* and *Partition* tools need to be invoked; subsequent tools need not be run. Unnecessary tool invocations can be avoided if the flow specification is modular.

   A number of design options are available at each step in the design process. For instance, the *Partition* tool can be either a manual partitioning tool, or an exact (but time consuming) tool such as CPLEX using integer linear programming techniques, or an efficient heuristic such as MIBS. 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 flow specification mechanism is thus compact and efficient.

2. Mechanisms to systematically track tool dependencies and automatically determine the sequence of tool invocations

   The user should not have to keep track of tools that have already run and those that need to be run. Also, 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. A mechanism that automatically determines the sequence of tool

possibilities for hardware synthesis: (1) Silage [Hilfinger85] code can be generated for the nodes mapped to hardware, and the generated code can be passed through the Mentor Graphics [MentorGraphics1] tools to generate a standard-cell implementation (2) VHDL code can be generated and passed through Synopsys [Synopsys] tools to generate a gate-array implementation. Similarly, different software synthesis strategies can be used; for instance, the software synthesis tool could generate C or assembly code. The interface generation depends on the desired architectural model. For instance, in a memory-mapped communication model, the address decoders and latches as well as the code that writes to and reads from the memory locations need to be synthesized.

There are several published approaches to the problem of synthesizing hardware and software from highlevel specifications. Typically, the complexity of the input to these synthesis systems corresponds to that of a single task in a system-level specification. Instead of reinventing the task-level synthesis tools, we have developed a system-level synthesis approach that builds upon them. The key aspects of this synthesis approach are discussed in Chapter 4.

Cosimulation

Once the hardware, software, and interface components are synthesized, the *Netlist Generator* puts them together and the system is simulated. Ptolemy [Buck94a] is a simulation environment that allows different models of computation to interact seamlessly. Due to its support for multiparadigm simulation, Ptolemy is suitable for simulating heterogeneous systems through the entire design process, from the specification to the implementation levels. The use of Ptolemy for hardware-software cosimulation is discussed in Chapter 4.

ferred implementation for some (say p) nodes, there are still a large number of design alternatives with respect to the remaining nodes ($(2*5)^{100-p}$). Determining the best design option for these remaining nodes is, in fact, a constrained optimization problem. In this thesis, we will focus on analytical tools to solve this problem, although the Design Assistant framework itself does not presuppose a particular tool.

In Chapter 3, we formulate the *extended partitioning* problem. Given the estimates for area and time values of each design option for all the tasks, we develop heuristics to determine the design options that result in a minimum overall area approximately, and still satisfy the throughput constraints. Heuristics are required because exact methods are computationally intensive. A heuristic, called MIBS (Mapping and Implementation Bin Selection), that solves the extended partitioning problem is presented in Chapter 3. MIBS is developed in two stages. First, in Section 3.3, an algorithm (Global Criticality/Local Phase, or GCLP) that solves just the hardware/software mapping and scheduling problem[1] is presented. GCLP is a very efficient heuristic; the complexity is $O(|N|^2)$, where $|N|$ is the number of tasks in the design specification. It uses a combination of global and local measures in order to determine the best mapping. The MIBS algorithm, described in Section 3.5, determines the mapping and implementation bin by a joint optimization process. It uses GCLP as a core and has cubic complexity in the order of number of tasks.

Cosynthesis

Once the application is partitioned into hardware and software, the individual hardware, software, and interface components are synthesized. The particular synthesis tool used depends on the desired technology. The figure shows just two

1. We call this the “binary partitioning problem”.

the desired throughput and the architectural model (maximum allowable hardware area, memory size, communication model, etc.). The *design flow* describes the sequence of tools that operate on the design data to generate the final implementation.

The components of the codesign flow include tools for estimation, partitioning, synthesis, and simulation.

Estimation

The *Estimation* tool generates estimates of the implementation metrics (area and execution time requirements) for each of the nodes in the graph in different implementations in hardware and software realizations. These estimates are used by the partitioning tool. Details of the estimation tool are discussed in Section 3.4.1 and Appendix A8.

Partitioning

After obtaining the estimates of the area and execution times, the next step in the codesign flow is *partitioning*. The goal of partitioning is to determine, for each task, three parameters: mapping (whether it is in hardware or software), schedule (*when* it executes, relative to other tasks), and implementation (which *type* of implementation option to use with respect to the algorithm, transformation, and area-time value). The Design Assistant framework allows the user to experiment with different partitioning tools. The figure shows three possible partitioning tools: manual, an ILP solver (CPLEX), or an efficient heuristic (MIBS).

Partitioning is a non-trivial problem. Consider a task-level specification, typically in the order of 50 to 100 nodes. Each task can be mapped to either hardware or software. Furthermore, within a given mapping, a task can be implemented in one of several options. Suppose there are 5 design options. Thus there are $(2*5)^{100}$ design options in the worst case! Although a designer may have a pre-

2.1 A Typical Codesign Flow and Tools for Codesign

A typical codesign flow is shown in Figure 2.2. A task-level specification (for example, a modem specified by a dataflow graph as in Figure 1.1) is transformed into the final implementation by a sequence of tools. The final implementation consists of custom and commodity programmable hardware components and the software running on the programmable components. The design objective is assumed to be to minimize the total hardware area. The design constraints include

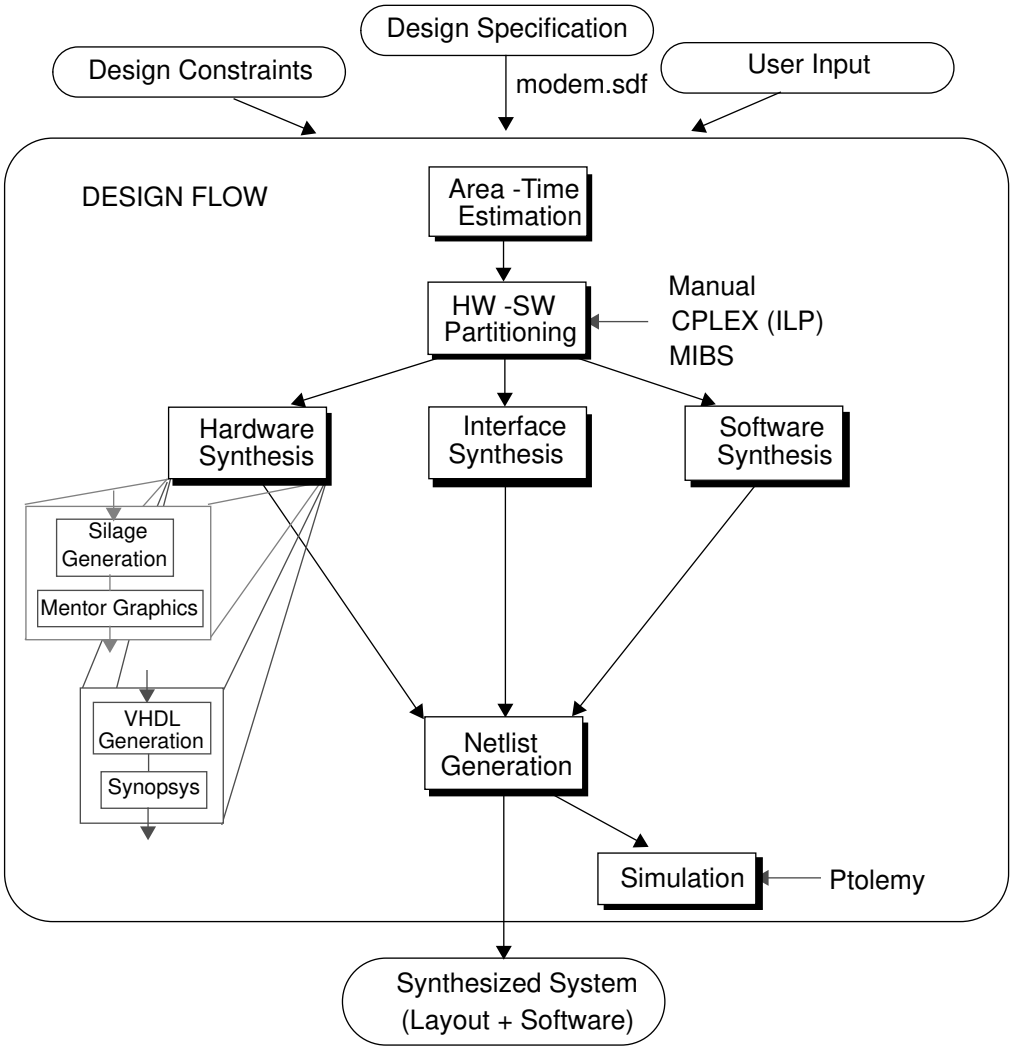

Figure 2.2. A Codesign Flow.

underlying design methodology management infrastructure for design-space exploration. The architecture of the Design Assistant is shown in Figure 2.1. The inputs to the Design Assistant include the design specification and the design constraints, and the output is an implementation for the system. The user configures the tools and constructs a design flow by determining the connectivity and the parameters of the tools. The design methodology manager is a transparent infrastructure that manages the design data, tools, and flow.

This chapter describes the various components of the Design Assistant. In Section 2.1, a typical codesign flow is discussed and the various tools required in the codesign process are outlined. In Section 2.2, the requirements of a design methodology management infrastructure are discussed. In Section 2.3, we outline the assumed specification semantics. The assumed target architecture is described in Section 2.4. Key restrictive assumptions made in the thesis are listed in Section 2.5.

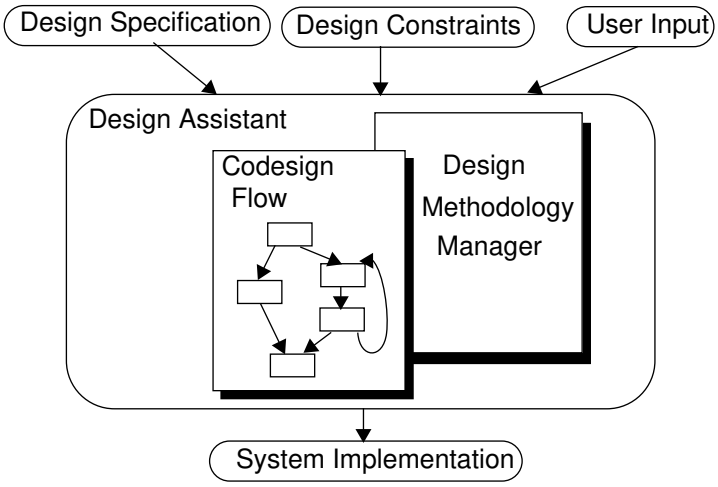

Figure 2.1. The Design Assistant

2

THE DESIGN ASSISTANT

Since system-level design oversees highlevel synthesis, logic synthesis, etc., decisions made at the system level impact all the layers below it. In other words, if the objective in system-level design is to come up with the "best" system implementation, there are a large number of design options. Each task can be represented by a multiplicity of algorithms, each algorithm selection can in turn be represented by several transformation-level and resource-level options. The system-level designer is faced with the question of selecting the best design option. In the context of mixed hardware-software systems, system-level design offers even greater challenges. How do we take advantage of the inherently distinct properties of hardware and software and still couple tightly their design processes? How can the mixed hardware-software system be simulated? In the light of the numerous tools involved in this process, can we relieve the designer of the burden of design management?

In order to address these issues, we present a framework called the *Design Assistant*. The Design Assistant provides a platform that spans the entire gamut of system-level design. It consists of: (1) specific tools for partitioning, synthesis, and simulation that are configured for a particular codesign flow by the user, and (2) an