A VHDL Synthesis Framework

Reid A. Baldwin, Sea Hawon Choi, and Moon Jung Chung
Department of Computer Science
Michigan State University
{baldwin,choi,chung}@cps.msu.edu

Abstract
We present a high-level synthesis framework which guides the designer in devising design methodologies. The framework supports backtracking and parallel exploration of alternative methodologies. Encoded knowledge assists the designer in selecting methodologies and tools, while designers retain control during the synthesis process. The framework facilitates maintainability and extendibility as new tools and methodologies are developed.

1 Introduction
Synthesizing VLSI circuits is time consuming and expensive. High level synthesis offers much more potential than the traditional method of entering schematics directly and then simulating and revising. Many researchers are discovering effective algorithms for pieces of the high level synthesis process and developing tools that work well some of the time. An effective synthesis system must integrate many of these tools in such a way that the appropriate algorithms can be applied to any circuit. How should such a system be organized?

Commercial software already exists that claims to perform high level synthesis. These systems restrict the set of language constructs in the behavioral description to those that map easily into structural elements. They usually work well in certain applications, but lock the user into predetermined algorithms which may produce low-quality output in other cases.

The most effective sequence of steps, called a methodology, to synthesize a circuit cannot be determined in advance. The high level synthesis process involves many tasks, such as scheduling operations to time steps, binding operations to functional units, synthesizing the control unit, etc. For many of these tasks, a variety of algorithms have been proposed, with no algorithm or tool being universally superior. Even the exact definitions of some of the tasks and the appropriate sequence are debatable. For example, some systems allocate functional units first and then schedule whereas others schedule first and then decide what functional units are needed.

Our approach to finding a methodology is to utilize information about individual tools to search for a sequence that accomplishes the overall task. A forward chaining search of this type is used in ADAM [10, 11]. Ulysses [3] uses a blackboard architecture to construct the methodology. Hierarchical planning, as used in [8] and our system, is more effective because knowledge about how to do abstract subtasks can be used more naturally.

Synthesis requires an iterative, trial and error approach. A designer chooses a particular approach and tries it, guessing critical parameters along the way. If it fails, he backtracks and tries something different until he finds an approach that succeeds. He uses his intuition and experience to determine which parameters to adjust or when to abandon his approach and try the tasks in a different order. Several existing synthesis systems attempt to operate as a black box with all user input contained in the input specification. This forces the designer to backtrack all the way to the beginning and adjust the system behavior with abstract control parameters. Seamless integration of user interaction and automated tools is very difficult to achieve.

Electronic CAD vendors and users recognize the importance of interoperable tools and created the CAD Frameworks Initiative (CFI) to facilitate the development of frameworks which integrate tools. CFI suggests standards for data interchange, intertool communication, and tool encapsulation. Since vendors are committed to interoperability, these standards will be widely observed in the future. CFI standards do not govern the selection of tools.

We have developed a framework for a high level synthesis system. Our framework guides the designer in choosing a methodology by suggesting and rating alternative ways of decomposing tasks. Backtracking when methodologies fail is supported. A unique feature of our framework is the ability to explore multiple methodologies in parallel. Our framework encapsulates knowledge about individual tools and tasks, facilitating maintenance and addition of new tools. Our control structure allows automated decisions where they are appropriate without impinging on the designer's authority. The encapsulated knowledge and the designer's own knowledge are used in concert to make design decisions.

In the next section, we present an overview of our framework. In the following section, we use an example to illustrate how our system may be used to synthesize a structural description from a VHDL behavioral description. Finally, we use another example to illustrate how new tools and task decompositions may be integrated into our system.
2 Framework Overview

Methodologies are developed by decomposing tasks hierarchically into simpler tasks until the individual tasks can be performed by single tool invocations. We call tool invocations terminal tasks and the abstract tasks that could conceivably be performed by different tools or combinations of tools non-terminal tasks. Methodologies are devised by hierarchically decomposing non-terminal tasks until all tasks are terminal.

We consider tool selection to be a special case of decomposition in which the set of subtasks is a single terminal task.

The architecture of our framework is illustrated in Figure 1. The designer interacts with a program called Cockpit, which keeps track of the current status and informs the user of possible actions. To assist the user in choosing an appropriate action, Cockpit interacts with several manager programs, which encapsulate design knowledge. The manager programs provide ratings for the possible task decompositions, invoke tools, and check the results. Manager programs must be maintained by tool integrators to reflect site specific information such as company design practices and different ways of installing tools. To assist tool integrators in creating manager programs, an additional utility uses information provided by tool vendors to automatically create source code for manager programs. A pre-processor is also provided to assist the tool integrator in maintaining the input file to Cockpit.

In order to compute intelligent ratings, the managers must often gather more information. This is accommodated by a query handling protocol. Managers can send queries to the managers of subtasks or parent tasks. Cockpit routes these query messages to the appropriate manager, but does not interpret them. We do not define which quantities may be queried; the implementor of each manager decides what queries to send and respond to.

2.1 Cockpit

Cockpit keeps track of the current state of the design process. Unlike the manager programs, Cockpit contains no task specific knowledge. Its information about the design process comes entirely from an input file indicating the set of possible tasks and what decompositions should be considered for each non-terminal task. In our prototype, we use a type of graph grammar to express this [2]. For each task, the input file indicates a manager program that encapsulates knowledge about that task.

Cockpit's display indicates to the user what design tasks have been done so far and what tasks remain. Cockpit determines what decompositions are available for remaining non-terminal tasks. It sends a message to the corresponding manager program requesting that the manager compute a rating of each decomposition's usefulness in the current situation. This information is displayed to the user.

Cockpit supports two modes of operation: manual mode and automatic mode. The manual mode is

Figure 1: Block Diagram of System
mally used for high level decisions. The designer may wish to use the automatic mode for the lower level decisions. In manual mode, Cockpit waits for the user to select a decomposition or execute a task. When the user selects a decomposition, Cockpit displays the new subtasks in place of the original task. When the user asks that a task be executed, Cockpit sends a message to the corresponding manager program. For terminal tasks, the manager responds by invoking the tool. The user invokes the automatic mode by executing a non-terminal task instead of selecting a decomposition. In response to an execute message for a non-terminal task, the manager uses encoded knowledge to select a decomposition and then execute the subtasks (also in automatic mode). If necessary, the designer may reverse any decision made by a manager program in automatic mode.

When the output of a task is not satisfactory, it is necessary to backtrack. Either different parameters must be supplied to some of the tools, different tools must be chosen, or a task must be decomposed in an entirely different way. The designer may request that certain task decompositions be reversed. Also, if a decomposition was requested by a manager program, that manager can direct Cockpit to reverse it. Cockpit saves the state of the session before backtracking in case the designer later changes his mind.

In automatic mode, it becomes advantageous to explore several alternatives simultaneously. (In manual mode, that would confuse even the best designers.) While the system explores some alternatives automatically, the designer can explore another. The ability to explore several methodologies in parallel is absent in other systems and would be very difficult to add to them.

2.2 Manager Programs

Manager programs are dynamic repositories of task specific knowledge. They must be upgraded as new tools are acquired and experience is gained. Each manager performs the following functions:

- Pre-Evaluation Assign a rating indicating the decomposition's likelihood of success in the current situation.

- Execution For terminal tasks, invoke whatever tools are needed. For non-terminal tasks, choose decompositions and issue messages to Cockpit requesting that subtasks be executed.

- Post-Evaluation Once the tools or subtasks have completed, check the appropriate constraints to determine whether the task was accomplished successfully.

- Query Handling Respond to task specific queries by estimating quantities or forwarding the query to a subtask or parent task. Implementors must negotiate with each other about what queries should be handled by each manager.

Implementors of manager programs may use whatever techniques they deem appropriate for these functions. For example, some managers may be algorithmic while others use rule bases or neural networks. We simply define a set of messages that managers are expected to be able to respond to or are allowed to send to Cockpit. A single manager program may encapsulate the knowledge for several tasks. Each message from Cockpit indicates what task is being evaluated or executed and provides the filenames for all of the inputs and outputs. The constraints may be included in one of the input files or may be passed to the manager by any other method the implementing programmer chooses.

2.3 Tool Encapsulation

Integrating tools can be difficult because every tool vendor has their own batch of command line options and environment variables which must be set correctly. The section of the manager program that invokes the tool must set the environment variables and command line options appropriately. Fortunately, CFI helps us address this problem. CFI proposes a Tool Encapsulation Specification (TES) language providing a uniform format for vendors to specify how to set environment variables, compose command lines, and interpret exit codes.

A TES file consists of four parts. The first part describes general tool information, such as tool name, path, and help string. The second part contains the tool's arguments. The next part describes the input/output data definition. Finally, the last part describes the structure of the command line, such as possible command arguments. It also provides information concerning the termination status codes which may be returned by the tool. Since these things depend on how the tool is being used, TES files usually contain statements, called getlntput statements, which request additional information. The command line and environment variables are determined using the responses to these requests.

When TES files are provided, the task of implementing the manager program can be simplified. A TES Compiler reads the TES file and generates C source code for a manager program, see Figure 1. The TES Compiler interacts with the tool integrator with a query for each getlntput statement. Some of the getlntput statements ask for information about how the tool is installed, such as pathnames for libraries. The tool integrator can simply answer these queries. Some getlntput statements, however, ask for information which will be different for each invocation, such as the names of input and output files. The tool integrator defers these queries. When source code is produced, a C preprocessor macro is inserted wherever the response would have been used. For each macro, the tool integrator must provide code that computes the appropriate response from information available in the message from Cockpit requesting invocation of the tool.

In addition to invoking tools and checking for successful completion, manager programs must compute ratings and answer queries. These functions are not addressed in TES files. For these functions, the TES Compiler simply inserts skeleton code which implements the interface with Cockpit. The tool integrator must encode the task specific knowledge.
3 Synthesis Example

In this section, we describe a synthesis scenario which illustrates how our framework is used. The tools and decompositions used are intended to be representative but certainly not unique. In the next section, we will discuss how additional tools and decompositions may be added to the system.

The circuit being synthesized is a controller for the internal combustion engine of a hybrid electric vehicle. The controller monitors engine rpm, battery voltage, current, fuel flow, etc. The primary output is a signal controlling the throttle. Several modes of operation are supported, such as adjusting the throttle to maintain battery charge, running the engine to re-charge the battery, and using battery power alone to eliminate emissions. The driver may select from a handful of parameters such as range on current battery charge, which the controller computes and sends in binary coded decimal format to a display unit. Since it is difficult to measure the battery charge directly, the controller maintains an estimate of charge by monitoring current and voltage and integrating the power used over time.

The objective is to design a multi-chip module (MCM) from a VHDL behavioral description of the controller. There are constraints on the number of connections between chips of the MCM and on area of each chip. There are also constraints on timing, the most serious being that the integration to estimate battery charge be called at proper intervals.

To start, we run Cockpit with an input file indicating the standard tools and task decompositions that are available on our site. The primary task, called MCM Synthesis, is initially the only icon displayed. Upon selecting this task, Cockpit tells us that it can be decomposed into the subtasks CDFG Generation, Scheduling and Allocation, Interconnect Synthesis, Control Synthesis, and Partitioning. We ask Cockpit to apply this decomposition and the MCM Synthesis icon is replaced in the display by the others.

3.1 CDFG Generation

Control and Data Flow Graph (CDFG) generation is the transformation of the VHDL behavioral description into a graph format encoding the data flow and control flow. The operations in the main process are labeled and all data and control dependences are located. Mutually exclusive operations are identified such that they may be scheduled on the same functional unit during the same control step.

When we click on the CDFG Generation icon, Cockpit tells us that there are two decompositions: a tool called CDFGen and another called ProcTrans. These tools differ in the way they handle procedure calls. CDFGen automatically expands the procedure body in place of the call. ProcTrans transforms the procedure body into a VHDL entity and treats the procedure call as a single operation to be performed by a functional unit synthesized from the new entity. ProcTrans allows us to take advantage of the fact that the behavior is divided into procedures to minimize communication, helping us satisfy pin constraints when we eventually partition into chips. However, the functional units synthesized from these new entities cannot share resources, so the resulting circuit will require more area.

Cockpit asks the appropriate manager to assign each a rating for the current situation. The rating for ProcTrans is high whenever the pin constraint is deemed difficult to satisfy and the area constraint is not considered crucial. At this point, there is very little information available about which constraints will be most important. ProcTrans gets a slightly higher rating since it succeeds more often overall. Although experienced designers may have their own opinions and ignore the ratings, we follow the system's advice and request ProcTrans. Cockpit replaces the CDF Gen icon with a ProcTrans icon. We invoke ProcTrans by selecting its icon and asking Cockpit to execute it. Cockpit sends a message to the manager program, which then invokes ProcTrans. This sequence of actions is illustrated in Figure 4.

3.2 Scheduling

Once CDFG generation has completed, we move on to Scheduling and Allocation. In this step, the number and type of functional units is determined and operations are assigned to control steps. Cockpit tells us that we have three decompositions available: ILP Scheduling followed by FU Binding, Force Directed Scheduling followed by FU Binding, or FU Allocation and Binding followed by List Scheduling. The first method, integer linear programming (ILP), guarantees that the sum of the areas of the functional units will be minimized subject to a constraint on the number of control steps. However, the computation time can be prohibitive. It gets a high rating only when there are both area and
Figure 4: Sequence of actions during CDFG generation

Figure 5: Decompositions of Allocation and Scheduling

timing constraints and the number of operations is small. Force directed scheduling does not guarantee an optimal area but usually comes close with much less computation than ILP. It gets a high rating when the most difficult constraint is the timing constraint. It is easiest to satisfy area constraints by choosing the functional units first and then using list scheduling. Since the timing constraints are more likely to be violated, this decomposition is rated high when the area constraints are deemed more crucial than the timing constraints.

When the pre-evaluation messages go to the scheduling manager (which managed all three decompositions), the manager needs to know which constraint is most crucial, so it sends queries to the parent task, MCM Synthesis. If we had previously backtracked, then the MCM Synthesis manager could forward the queries to the managers of the appropriate failed subtasks. For example, if we had previously gotten to Partitioning and then failed, the manager of the failed process could probably give a useful estimate of area and delay. In this case, MCM Synthesis forwards the query to the manager of the only completed task, CDFG Generation. Since the tool chosen for CDFG Generation typically results in large circuits, the area constraint is deemed most crucial. Obviously, the intelligent handling of these queries required task specific knowledge. The logic for responding to a query or determining how to forward it would likely be very different for other tasks and other queries. After receiving the reply from the query, the scheduling manager rates the third decomposition highest and the user selects it. This sequence of actions is illustrated in Figure 6. Particular tools were then selected and executed for FU Allocation and Binding and List Scheduling.

At this point in our scenario, we run into a problem. The schedule produced requires an unacceptable number of control steps. To solve this problem, we must reverse some of our decisions and try something else. We could just back up to the FU Allocation and Binding task, selecting a different set of functional units that might work better. In this case, our engineering judgement tells us that the underlying problem is the decision we made back in CDFG generation. We backtrack by selecting the ProcTrans icon and then pushing the button labeled “backtrack.” Cockpit returns the display to the state just before we selected ProcTrans and redisplay our choices for the task CDFG Generation. This time we choose CDFGGen.
3. Cockpit routes query to MCM Synth manager.
4. MCM Synth. manager forwards query to manager of CDFG Generation task.
5. Cockpit routes query to CDFG Generation manager.
6. CDFG Generation manager replies.
7. Cockpit routes reply to original requester.
8. Scheduling manager returns ratings.

Figure 6: Query handling actions during scheduling

We repeat the scheduling and allocation steps as before (this time with better estimates) and produce an acceptable schedule.

3.3 Interconnect Synthesis

Since we trust the knowledge encoded for Interconnect Synthesis, we use the automatic mode. We select the Interconnect Synthesis icon and Cockpit displays our only choice. It tells us that the task can be decomposed into Register Assignment, Operator Assignment, and Schematic Generation. We execute the task in automatic mode by clicking the "execute" button. Cockpit sends an execute message to the manager for Interconnect Synthesis.

Figure 7: Decomposition of Interconnect Synthesis

Figure 8: Decompositions of Schematic Generation

The execute function of that manager sends messages back requesting that the task be decomposed into the three subtasks and then that the Register Assignment task be executed. Cockpit sends an execute message to the Register Assignment manager, which chooses a tool and sends back a message requesting that the tool be executed. Cockpit sends a message to the tool's manager which invokes the tool and sends back a reply that it succeeded. The Interconnect Synthesis manager executes the Operator Assignment and Schematic Generation tasks in the same fashion.

The Schematic Generation task has two decompositions, one using only multiplexors to connect registers to functional units and the other using busses. The multiplexor only method usually results in a faster circuit that requires more area. Thus, the relative ratings depend on the area and timing constraints. The execute function for Schematic Generation uses the simple but effective strategy of trying the higher rated decomposition first and then invoking the other if it fails. A more advanced strategy might investigate both decompositions in parallel if the ratings are both low enough that success is in doubt. In this case, the multiplexor based schematic is attempted first and it
meets the overall timing and area constraints. The sequence of messages to make these decisions is illustrated in Figure 9.

Although decisions about interconnect synthesis were made automatically by the system, the designer can decide to reverse any of those decisions. This became necessary when we proceeded to the Partitioning task. We were unable to partition the circuit into chips while satisfying the pin constraints. In this case, we decided to backtrack to the Schematic Generation task and attempt the bus based solution. That solution satisfied the timing and area constraints and, when Partitioning was repeated, resulted in an acceptable number of pins per chip.

4 Tool Integration Example

When a new tool is obtained from a vendor, it must be integrated into the system so that it is available to designers. As an example, we illustrate how asimut, a logic level simulation tool from the Alliance tool set, could be integrated into our framework. Tool integration is likely to be performed by a system administrator and the process may be completely hidden from designers. A portion of the TES file for asimut is shown in Figure 10.

We use a utility called the TES Compiler to get information from the TES file and tool installer and create the manager program for the tool. Part of the interaction between the TES Compiler and the tool installer is shown in Figure 11. The TES

```
... (argumentList
  (argument root STRING
  (getInput (Label "Root")))
 })

  (argument root_description BOOLEAN
    (ifTrue "-b")
    (getInput (Label "Root description")
      (default (false))
      (description "The root description"
        "is a behavioral description")
    )
  )
...`

Figure 10: Portion of TES Specification
Compiler queries the tool installer once for each `getInput` statement in the TES file. When asked which directories to search for "MBK.CATA_LIB", the tool installer replies that the current directory and `/home/alliance-1.1/cells/sclib` should be searched. However, much of the information requested will not be available until the tool is invoked. When queried for "VH_MAXERR", the tool installer defers. When the TES Compiler produces source code for the manager program, any deferred values will be treated as pre-processor macros. Before compiling the manager program, the tool integrator must define these macros indicating how the value should be computed from information available to the manager at invocation time. A portion of the source produced is shown in Figure 12.

The tool integrator must indicate what tasks the tool performs. This is done by adding the decomposition shown in Figure 13 to the input file of Cockpit. If the non-terminal task `Logic Simulation` is not already used as a subtask in other decompositions, then extra decompositions, such as the one shown in Figure 14, must be added which indicate how the task `Logic Simulation` contributes to a higher level task. The tool integrator must also add code for the pre-evaluation function, which rates how well `asimut` can do `Logic Simulation`. Also, if parent tasks are likely to send queries, code must be added to respond to them appropriately.

5 Conclusion

We have implemented a prototype high level synthesis framework using the architecture described. The prototype system includes the Cockpit program and several manager programs. We are currently integrating more tools and improving the encapsulated knowledge.

Our framework facilitates the use of encapsulated control knowledge to select methodologies and tools while not infringing the designer's authority. The encapsulated knowledge is made available to the user by rating the available task decompositions. When the user trusts the encapsulated knowledge, tasks may be run in an automatic mode where manager programs make decisions about methodology instead of simply offering advice. The framework supports backtracking to change methodology and exploring multiple methodologies in parallel.

By organizing knowledge about how to decompose tasks and invoke tools into a collection of manager programs, we make this knowledge have maintainable and extendible. When tool vendors supply TES files, the process of integrating a new tool into our system can be automated.

```
%tes asimut.tes
Compiling "asimut".
getInput "Root"
  1) Set value.
  2) Defer value.
> 2
getInput "Pattern"
  1) Set value.
  2) Defer value.
> 2
...
getInput "MBK_CATA_LIB"
  1) Set value.
  2) Defer value.
> 1
value>/home/alliance-1.1/cells/sclib
value>...
getInput "VH_MAXERR"
  1) Set value.
  2) Defer value.
> 2
Compilation of "asimut" done.
```

Figure 11: asimut TES Compilation
```c
#define ROOT /* to be supplied */
...
#define PATTERN /* to be supplied */
#define VHMAXERR /* to be supplied */
...

execute()
{
    /* A context is created for the tool. */
    context = cfitcContextCreate(&error);
    cfitcContextSetTooltype(context, "alliance", &error);
    ...
    /* Create an instance of the tool. */
    tool = cfitcToolCreate(&error);
    /* Set the TES parameters. */
    cfitcToolSetCommand(tool, "asimut", &error);
    cfitcToolSetHostname(tool, "golden", &error);
    cfitcToolSetContext(tool, context, &error);
    cfitcBaseSetPropInt(tool, "VH_MAXERR", VHMAXERR, &error);
    cfitcBaseSetPropString(tool, "MBK_CATA_LIB".
        "/home/alliance-1.1/cells/slib", &error);
    ...
    /* Start the tool. */
    cfitcToolStart(tool, &error);
}
```

Figure 12: Manager Source Code produced by TES Compiler

Figure 13: Decomposition of Logic Simulation

Figure 14: Decomposition of MCM Design
References


