A Compiler for HP VEE

Steven Greenbaum

Stanley Jefferson

With the addition of a compiler, HP VEE programs can now benefit from improved execution speed and still provide the advantages of an interactive interpreter.

This article presents the major algorithmic aspects of a compiler for the Hewlett-Packard Visual Engineering Environment (HP VEE). HP VEE is a powerful visual programming language that simplifies the development of engineering test-and-measurement software. In the HP VEE development environment, engineers design programs by linking visual objects (also called devices) into block diagrams. Features provided in HP VEE include:

- Support for engineering math and graphics
- Instrument control
- Concurrency
- Data management
- GUI support
- Test sequencing
- Interactive development and debugging environment.

Beginning with release 4.0, HP VEE uses a compiler to improve the execution speed of programs. The compiler translates an HP VEE program into byte-code that is executed by an efficient interpreter embedded in HP VEE. By analyzing the control structures and data type use of an HP VEE program, the compiler determines the evaluation order of devices, eliminates unnecessary run-time decisions, and uses appropriate data structures.

The HP VEE 4.0 compiler increases the performance of computation-intensive programs by about 40 times over previous versions of HP VEE. In applications where execution speed is constrained by instruments, file input and output, or display update, performance typically increases by 150 to 400 percent.

Steven Greenbaum

A member of the technical staff at HP Laboratories since 1989, Steve Greenbaum is currently researching "hardware-in-the-loop" systems and programming for distributed systems. He has a PhD degree in computer science (1986) from the University of Illinois at Urbana-Champaign and a BS degree in computer science (1980) from Syracuse University. Steve was born in New York City, is married, and has two children. In his leisure time he enjoys playing guitar and taking field trips with his family.

Stanley Jefferson

Stanley Jefferson is a member of the technical staff at HP Laboratories, where he began his career at HP in 1990. He is currently doing research in the area of "hardware-in-the-loop" systems. He has a PhD degree in computer science (1988) from the University of Illinois at Urbana-Champaign. He received BS (1977) and MA (1979) degrees in mathematics from the University of California at Davis. Stan was born in Oakland, California, is married, and has two children. He enjoys playing piano and day trips to the beach with his family.

The compiler described in this article is a prototype developed by HP Laboratories to compile HP VEE 3.2 programs. The compiler in HP VEE 4.0 differs in some details.The HP VEE prototype compiler consists of five components:

- Graph Transformation. Transformations are performed on a graph representation of the HP VEE program. The transformations facilitate future compilation phases.
- Device Scheduling. An execution ordering of devices is obtained. The ordering may have hierarchical elements, such as iterators, that are recursively ordered. The ordering preserves the data flow and control flow relationships among devices in the HP VEE program. Scheduling does not, however, represent the run-time flow branching behavior of special devices such as If/Then/Else.
- Guard Assignment. The structure produced by scheduling is extended with constructs that represent run-time flow branching. Each device is annotated with boolean guards that represent conditions that must be satisfied at run time for the device to run. Adjacent devices with similar guards are grouped together to decrease redundancy of run-time guard processing. Guards can result from explicit HP VEE branching constructs such as If/Then/Else, or they can result from implicit properties of other devices, such as guards that indicate whether an iterator has run at least once.
- Type Annotation. Devices are annotated with type information that gives a conservative analysis of what types of data are input to, and output from, a device. The annotations can be used to generate type-specific code.
- Code Generation. The data structures maintained by the compiler are traversed to generate target code. The prototype compiler can generate C code and byte-code. However, code generation is relatively straightforward to implement for most target languages.

To simplify the presentation, many aspects of the HP VEE language and compiler are omitted or given cursory treatment. Notable in this regard is our cursory treatment of concurrency mechanisms (both the prototype compiler and the HP VEE 4.0 compiler handle concurrency). Only a brief, somewhat formal description of the HP VEE language is presented. Less formal descriptions of HP VEE are given in the HP VEE manuals.[1,2]

Semantic Overview

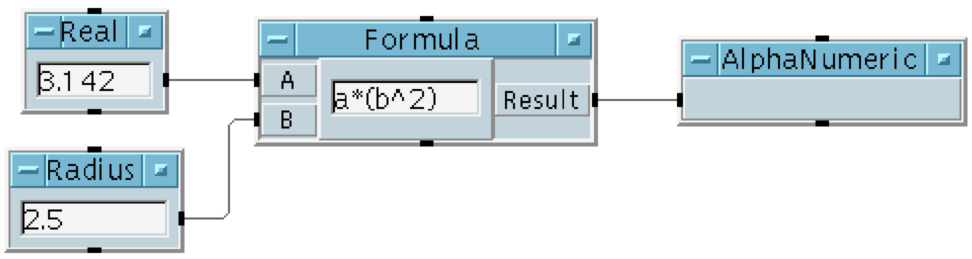
HP VEE programs are constructed by connecting devices together to form block diagrams. A simple HP VEE program is displayed in **Figure 1**. HP VEE has an extensive collection of built-in devices. Iterator, junction, and conditional devices affect program control flow. Other devices manipulate data or perform side-effects (some devices do both). Another set of devices are available for applied mathematics, controlling instruments, displaying engineering graphs, building user interfaces, data management, and performing I/O.

Pins and Devices

Devices may have any number of input and output pins, depending on the device's function. Connections can be made from output pins to input pins to route data or control signals between devices. Several connection lines can emanate from a single output pin, but at most one connection line can be attached to an input pin. A pin is considered to be connected if there is a connection between it and another pin. Unconnected pins are not necessarily an

Figure 1

A simple HP VEE program to compute the area of a circle.



error condition, but they serve no useful purpose. It will simplify discussions to assume that they are not present. Thus, the statement "data is placed on all output pins" implicitly excludes all unconnected output pins.

When a device executes, it performs a computation based on the values present on its input pins (if any) and produces results that are placed on appropriate output pins (if any). The value placed on an output pin is also propagated to any input pins that are connected to it. The combination of placing a value on an output pin and propagating the value is called "firing the output pin." Only one value can be present on a pin, so previous values are overwritten by new values. Unlike traditional data-flow models where input values are consumed, in HP VEE values on data input pins remain available for further use after they are used as input by an executing device.

There are five kinds of pins that may be attached to a device:

- *Data pins* provide the input/output interface to a device. The data input pins attach to the left edge of a device and the data output pins attach to the right edge of a device. Most devices will not operate until data is present at all data input pins. After a device operates, data is placed on the output data pins.
- *Sequence pins* are an option that allow greater control over the order in which devices operate. Most devices have sequence input and sequence output pins. The sequence input pin is attached to the middle of the top edge of a device and the sequence output pin is attached to the middle of the bottom edge of a device. All of the devices in **Figure 1** have unconnected sequence pins. Either sequence pin may be left unconnected. If the sequence input of a device is connected, then the device will not operate until the data input pins and sequence input pin have data. Sequence output pins are explained later.
- *Execute pins* are special input pins that force the device to operate and place results on its output pins. Execute pins are usually referred to as XEQ pins. XEQ pins operate regardless of the presence of other inputs.
- *Control pins* are special inputs that affect the internal state of a device, but have no effect on the propagation of data values through the device. Common control pins are Clear and Reset. Control pins operate regardless of the presence of other inputs.
- *Error pins* are optional output pins. The presence of an error pin causes any errors generated by the attached device to be trapped. The appropriate error code is output on the error pin.

A device can also have individual properties that are specified at development time. For example, the buffer size and grid type can be specified for a strip chart display device.

Data Types

The data types in HP VEE are integer, real, complex, polar complex, waveform, spectrum, coordinate, enum, text, and record. Multidimensional arrays can be built from these data types. Generally, the input and output pins on a device are not typed, and connections are never typed. Rather, the data objects themselves are typed. Most of the devices in HP VEE will accept any type of data, and they automatically perform any necessary type conversions. For example, the addition device will accept any combination of integer, real, complex, or array arguments. Appropriate types are output based on the input types. Some devices require particular data types as input and will either perform a conversion or signal an error when presented with a data object not meeting the type requirement. Most devices allow a user-specified type conversion to be associated with each input pin. In addition, most devices allow the user to require that the data be a certain shape (scalar, array, one-dimensional array, two-dimensional array, etc.). There is a nil value that is a value of every type and means "no information." The absence of a value at a pin is different from the presence of a nil value.

Terminology

A *connection* is a set (unordered) consisting of an input pin and an output pin. Devices x and y are *connected* if there is a connection c such that one of the pins in c is attached to device x and the other pin in c is attached to device y. Unless otherwise specified, connections between devices are undirected. Hence, a connection between devices x and y is also a connection between devices y and x. If $x_1,...,x_n$ is a sequence of devices and $c_1,...,c_{n-1}$ is a sequence of connections such that c_i is a connection between x_i and x_{i+1} for $i = 1,...,n-1$, we say that $c_1,...,c_{n-1}$ is an *(undirected) path* from x_1 to x_n. For example, in **Figure 1** there is a path from the Radius device to the Real device. A diagram is *pathwise connected* if there is at least one path between every pair of devices in the diagram. A device x is a *direct ancestor* of a device y if there is a connection from an output pin of x to an input pin of y. In this case, we also say that y is a *direct descendant* of x. A device u is an *ancestor* of a device v if there is a sequence of devices $w_1,...,w_n$ with $u = w_1$ and $v = w_n$ such that w_i is a direct ancestor of w_{i+1} for $i = 1,...,n - 1$. Also, if c_i is a connection from an output pin of w_i to an input pin of w_{i+1}, then the sequence $c_1,...,c_{n-1}$ is called a *directed path* from u to v. If u is an ancestor of v, we may also say v is a descendant of u. A *cycle* or *feedback loop* is a directed path from a device to itself. A pin p *occurs* in a cycle $c_1,...,c_m$ if p is a member of some c_i. The *descendants of a device* u are all the devices v for which a directed path exists from u to v. In a diagram with a cycle, it is possible for a device to be a member of its descendants.

Data Flow

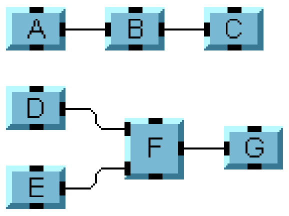
An HP VEE program is run by executing the devices in the program. The order in which the devices execute is constrained by the connections between the devices and some built-in priority rules. We say that a device's *data dependencies are satisfied* if all of its connected data input pins and its sequence input pin (if connected) have data present. The basic rule governing the execution order of devices is that a device can execute only when its data dependencies are satisfied. We call this the *data dependency rule.*

A device that is neither an iterator, junction, nor asynchronous* device is called a *primitive device.* The execution order of primitive devices in a connected diagram where the only connections are between data output pins and data input or sequence input pins is governed by the data dependency rule. A diagram of this form is run by allowing each device to execute at most once, subject to the ordering constraint imposed by the data dependency rule. One way to run such a diagram is to repeatedly choose an unexecuted device whose data dependencies are satisfied and execute it until there are no devices left to choose. Each time a device executes, the values propagated from its outputs may satisfy the data dependencies of descendant devices, thus making additional devices available for execution. The process of running a diagram (or subdiagram) D is referred to as a sweep over D. The program in **Figure 1** can execute its devices in the order Radius, Real, Formula, Alphanumeric or in the order Real, Radius, Formula, Alphanumeric.

* Asynchronous devices, such as Delay and Confirm, are not treated in this paper.

Figure 2

A diagram with two independent threads.



A maximal pathwise connected subdiagram of a diagram D is called an *independent thread* of D (connections to control pins are ignored when determining independent threads in HP VEE). For example, the diagram in **Figure 2** has two independent threads. Suppose that D is an HP VEE program consisting of n independent threads. Then a sweep over D initiates a subsweep over each of the n independent threads. Each subsweep executes independently of the others and the n subsweeps all run concurrently (or in a time-sliced manner). The sweep over D is completed when all n subsweeps are completed.

Later, we will present control constructs that initiate subsweeps over subdiagrams. In general, it is possible to have arbitrarily nested subsweeps during the execution of a program. Let $s_1,...,s_n$ be a sequence of sweeps such that s_{i+1} is a subsweep of s_i for $i = 1,...,n - 1$. Then, the sequence $s_1,...,s_n$ is called a *nested sequence of sweeps*, and we say that s_i is a *supersweep* of s_j whenever $i < j$. If $s_1,...,s_n$ is a nested sequence of sweeps and we are currently executing the sweep s_n, then $s_1,...,s_n$ are said to be *active*, but only s_n is said to be executing.

The following *execution nesting rule* is a generalization of the rule that devices execute at most once per sweep. Let $s_1,...,s_n$ be a nested sequence of sweeps and d be any device except a *junction* device. If d is directly executed by the innermost subsweep s_n, then device d can not be directly executed again by any of the s_i, for $i \leq n$, but it may be directly executed again by a newly created subsweep of an s_i for $i < n$.

We glossed over a technical detail regarding device execution. Consider a device that has a data input pin, an XEQ pin, and a control pin. Each of these pins corresponds to a different action. Which actions qualify as executing the device? The detailed answer is given in the section "Execute and Control Pin Splitting" on page 107. The short answer in this particular case is that the device is viewed as three devices where each device corresponds to a different input pin, and each of the three devices can execute subject to the execution nesting rule.

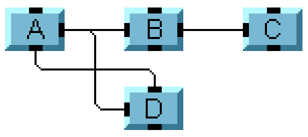
Sequence Pins

In some cases data dependencies in HP VEE are not capable of adequately specifying the order of execution of devices. Sequence output pins provide an additional mechanism for constraining the execution order. The intuitive idea behind sequence output pins is that the sequence output pin of a device d fires after d has executed and only when no further execution is possible in the devices descended from the data and error output pins of d. Informally, the sequence output pin on device d attempts to capture the notion of completing the future computation descended from d. In the example shown in **Figure 3** the sequence out pin of device A does not fire until A, B, and C have executed. It follows that the devices can only execute in the order A, B, C, and D.

Because of cycles, descendants of a device d may include devices that executed before d. Devices that executed before d cannot reasonably be considered part of the future computation descended from d. Therefore, if we are in the midst of a sweep that has executed device *d*, we define the *future computation descended from d* as the descendants of the data and error output pins of d that did not execute before d in any currently active sweep.

Figure 3

A sequence pin example. The only possible order of execution is A, B, C, *and* D. *Note that the lines do not connect where they cross.*



A couple of issues complicate the determination of when to fire a sequence output pin. The following must be ensured before firing the sequence output pin of any executed device d:

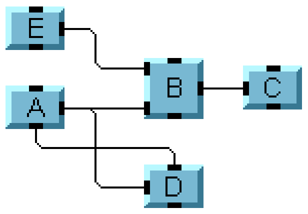
1. The execution of devices in the future computation descended from d has proceeded as far as possible.
2. There are no executed devices in the future computation descended from d having unfired sequence output pins.

Determining (1) is complicated by the possible presence of devices in the future computation descended from d that depend on data coming from devices not descended from d. This is illustrated in **Figure 4**. A very coarse-grained but simple way to ensure (1) is to reach the point, after evaluating d, where the execution of devices in the current sweep has proceeded as far as possible without firing a sequence pin. Once this point is reached, a sequence output pin can be chosen so that (2) is ensured. One way to ensure (2) is to choose the most recently executed device with an unfired sequence output pin and fire that device's sequence output pin. It is possible to fire sequence pins more aggressively and still ensure (1) and (2), but it requires deeper analysis.

We now give a more detailed account of sequence pin firing in HP VEE 3.2. Consider a sweep (or subsweep) over a diagram D that has run to the point where no device in D is executing and no device in D is capable of executing without firing a sequence output pin. Choose an executed

Figure 4

The future computation descended from A *depends on data from* E. *Devices* A, E, B, *and* C *must execute before the sequence output pin on* A *fires.*



device d in D such that d is the most recently executed device having an unfired sequence output pin (if there are none then the sweep is finished). Propagate a nil value from the sequence output pin of d, and continue the sweep of D until no unexecuted devices with satisfied data dependencies are available. Repeat this process of firing sequence output pins until all sequence output pins on executed devices in D have been fired. When further repetitions are not possible, the sweep over D is completed.

Note that sequence output pins in different subsweeps are handled independently. Sequence output pins are fired as part of the subsweep that executed their attached device, and the order of their firing depends only on the devices and sequence output pins in that subsweep.

If/Then/Else

The If/Then/Else device is used for branching the data flow, and hence the execution flow. The programmer specifies any number of named data input pins and a list of expressions over those pins. Each expression corresponds to a different data output pin. When the If/Then/Else device executes, the expressions are evaluated in order until the first expression that evaluates to a True (nonzero) value is found. At this point, expression evaluation ceases, and the value is output on the data output pin corresponding to the true expression. If no expression evaluates to True, then zero is output on the default Else data output pin. Note that only one data output pin of the If/Then/Else device outputs a value. Thus, the other data output pins will not propagate a value, and hence none of their descendants will be able to execute (because of the data dependency rule).

Iterators

One consequence of the execution nesting rule is that feedback loops in HP VEE can never result in iteration. Instead, HP VEE has iterator devices that explicitly perform iteration. These iterator devices repeatedly run the subdiagram descended from their output pin. It should be noted that iterator devices do not impose any hierarchical structure on the displayed HP VEE program, but are simply displayed as an ordinary device. The ForCount iterator is shown in **Figure 5**. The ForCount device outputs the sequence 0, 1, 2,...,n – 1, where n is an iteration count that can be input at runtime or set at development time. When the ForCount is run during a sweep, it outputs a zero (assuming the iteration count is greater than zero) and starts a subsweep on the diagram descended from its output pin. When the subsweep is finished, the ForCount determines whether it has output its last value. If not, the ForCount outputs its next value and performs another subsweep. This process of performing subsweeps continues until the final value of the ForCount is output, at which time the ForCount ends its execution. Note that the ForCount is in an outer sweep relative to the subsweeps on its descendants, and that it is executed only once during the sweep that initiated its execution (one can think of the ForCount as being in a paused state when its subsweeps run).

The following *active data rule* applies to devices such as iterators, junctions, and UserObjects that can create a new subsweep. Let d be a device that creates a new subsweep. Immediately before each subsweep created by d, all data created in the previous subsweep (if there was one) performed by d is inactivated (that is, made unusable). This

Figure 5

An example using ForCount*. The result of running the program is shown. The* Formula *and* Logging AlphaNumeric *devices have been iterated five times.*

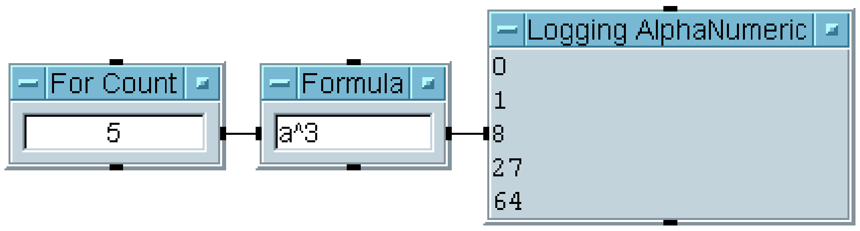
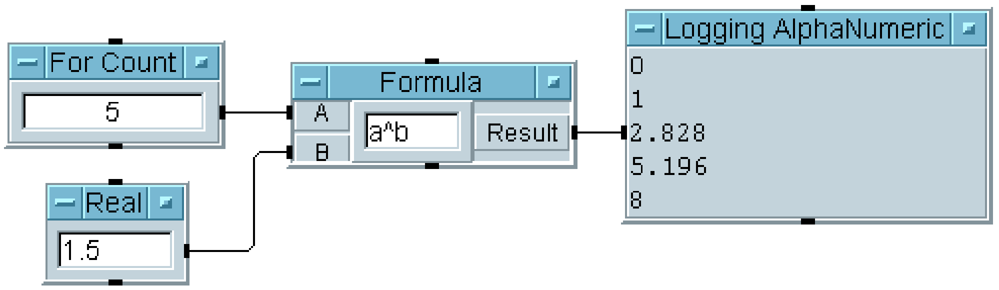


Figure 6

This example, shown after execution, illustrates that data created outside the ForCount*'s subsweeps, by the* Real *constant device, remains active.*



includes data created in subsweeps that are nested inside the subsweep performed by d. To lend a more intuitive feel to feedback loops, certain values occurring in feedback loops are exempted from inactivation. Specifically, if an input pin occurs in a cycle and the data value on that pin was not used in the sweep it was generated in, then the data value on that pin is not inactivated (since the intent of the feedback loop was to use this value on the next iteration).

Without the active data rule, the data dependencies of devices could be trivially satisfied on subsequent subsweeps. The active data rule is illustrated in **Figure 6** and **Figure 7**.

Besides ForCount, there are other iterator devices that work similarly because they output values and repeatedly perform subsweeps on their descendant subdiagrams until a termination condition is reached. Iterator devices can occur as descendants of iterator devices to any depth. This will result in nested subsweeps. If two iterator devices are in the same sweep, we will assume that the subdiagrams descended from any of the output pins (including sequence and error outputs) of one device do not intersect with the subdiagrams descended from any of the output pins of the other device.* All iterator devices in the same innermost sweep execute as a group, concurrently or in a time-sliced manner, so that no iterator is starved by another iterator performing a large (or unbounded) number of iterations.

Junctions

The junction device allows data from two or more data input pins to be merged into a single data output pin. This is useful for merging the branches of an If/Then/Else back

* In VEE 3.2 intersection was allowed, but the meaning was not well-defined. In VEE 4.0 iterator intersection is signaled as an error.

Figure 7

This example, shown after execution, illustrates that the data placed on the AlphaNumeric*'s data input pin on the next-to-last iteration was inactivated at the start of the last iteration. Otherwise, the* AlphaNumeric *would have displayed the data. Note that the last value output by the* ForCount *is 9.*

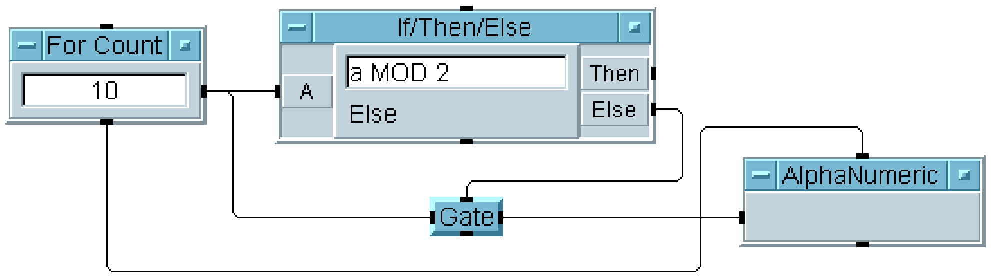
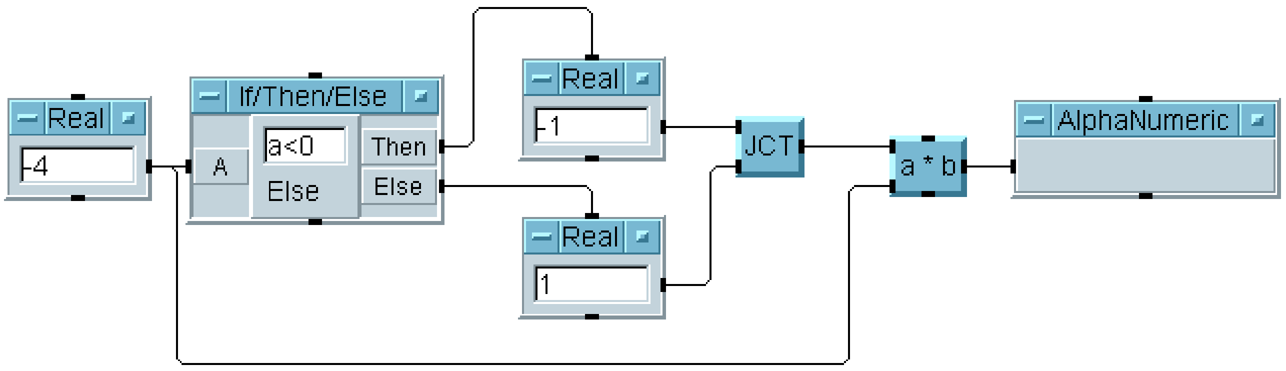


Figure 8

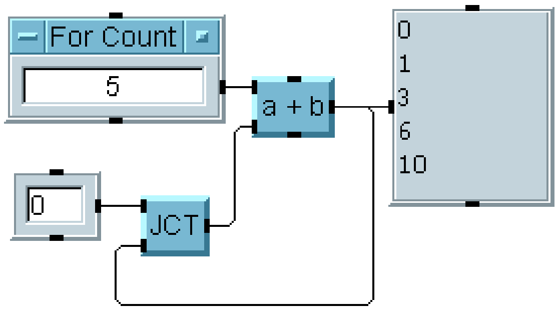
An example of a junction merging the branches of an If/Then/Else*. When run, this program displays the absolute value of its input. In this case, the* AlphaNumeric *would display 4 when the program is run.*



together, for initializing feedback loops, or for iterating the inputs of the junction. See **Figure 8** and **Figure 9** for examples. In the If/Then/Else merge and feedback initialization cases, at most one of the junction device's data input pins will receive a value during a sweep. Thus, the junction device does not have to satisfy the data dependency rule to execute. The junction device can execute whenever one or more data input pins has a value. If a junction, j, initiates a sweep s of its descendants, then neither s nor any subsweep nested inside s may execute j. This restriction prevents iteration resulting from feedback. Unlike other devices, the junction device consumes its data input values when it executes. When the junction device executes, it repeatedly sweeps over the descendants of its data output pin just like an iterator, using the data input values as data output values. Thus, subsweeps performed by a junction are subject to the active data rule. Although a junction performs subsweeps in the same manner as an iterator, it is not an iterator. The prototype compiler assumes that junctions that are in the same sweep execute in an arbitrary serial order rather than concurrently, and the descendants of different junctions in the same sweep are allowed to intersect.

Figure 9

This example, shown after execution, illustrates a junction being used to initialize a feedback loop.

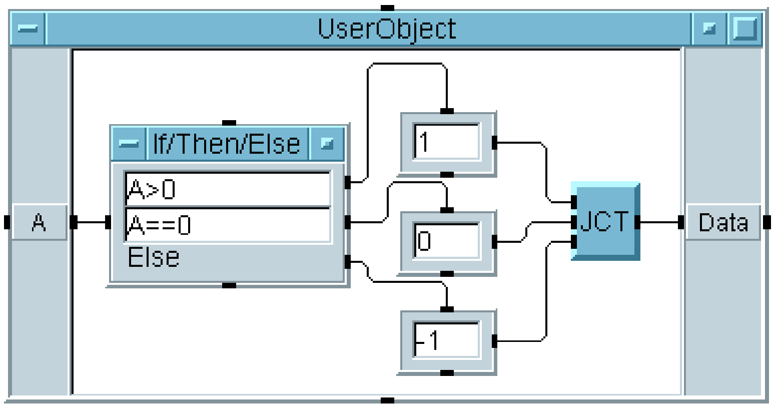


User Objects

Subprograms can be written in HP VEE using the UserObject device. The UserObject provides a subwindow in which a block diagram can be constructed. A UserObject is shown in **Figure 10**. The data input pins of a UserObject have corresponding terminals inside the UserObject's subwindow. The diagram contained in the UserObject can connect to these terminals in order to obtain the data on the UserObject's data input pins. The data output pins of a UserObject are handled similarly. UserObjects operate under the same rules as any primitive device. All data inputs must be present before the UserObject executes. When the UserObject executes, its diagram is executed as if it were a top-level diagram. The sweep over the UserObject's diagram runs further subsweeps over the independent threads of the diagram. When the sweep of the UserObject's diagram is finished, the last values placed on the terminals of the UserObject's data output pins are transferred to their corresponding data output pins, and the UserObject completes its execution. UserObjects may also have error and sequence pin connections.

Figure 10

A UserObject *that determines the sign of its input.*



Phases of a Sweep

During a sweep s over a diagram D, a priority ordering affects the order of device executions and sequence output pin firings in D. If a subsweep of s is initiated, then its priority ordering will begin fresh and will be independent of the ordering in sweep s. The following summarizes the priority classes of a sweep from high to low priority.

1. If there are primitive devices whose data dependencies are satisfied, then one is chosen and executed. This process is continued until there are no primitive devices whose dependencies are satisfied. Note that additional primitive devices may have their data dependencies satisfied as data is propagated from the output pins of executed devices, and thus they will become eligible for execution during this priority phase.
2. All junction devices that have data present on at least one input pin are executed. The junctions are executed in an arbitrary serial order, and each one initiates a new subsweep.
3. All iterator devices that have their data dependencies satisfied are executed. The iterator devices are executed concurrently and each one initiates a new subsweep.
4. If a sequence output pin is eligible to fire, it is fired as described earlier, and then the current sweep continues at step 1. If there are no sequence output pins eligible to fire, the current sweep is over.

Example

Consider the execution of the program in **Figure 9**. The zero constant device executes first as it is the only primitive device with its data dependencies satisfied. Then the junction device executes, outputting the data that is on its upper data input pin. At this point there are no primitives or junctions that can execute, so we execute the ForCount. The ForCount outputs a zero and starts a subsweep over its descendants. The subsweep begins with the highest priority devices. The addition device is a primitive device and now has its data dependencies satisfied, so it executes. The data output by the addition device satisfies the data dependencies of the display device and the junction. Since the display device is primitive it has priority over the junction, so it executes, displaying a zero. There are now no primitive devices left in the subsweep that can execute, so the junction executes. The junction outputs the data that is on its bottom input pin and starts a subsweep. The junction's subsweep immediately terminates since all of the descendants of the junction are prohibited from executing because of the execution nesting rule. No more devices can execute in the subsweep started by the ForCount since all of the devices descended from the ForCount have executed once in this current subsweep. Thus, the current subsweep ends and the ForCount readies for another iteration. Before performing the next iteration, the active data rule is applied, inactivating all of the data created on the previous sweep except the data that was output by the junction, since it is exempt because of feedback. The ForCount outputs a one and performs another sweep. The descendants of the ForCount are all eligible to run since we have backed out of the subsweep in which they were previously run. Thus, this second subsweep proceeds in the same manner as the previous subsweep. Iterations continue in this way until the ForCount has performed five iterations, at which point the top-level sweep is terminated. Note that feedback is simply a mechanism for using values generated in previous sweeps.

Architecture of the Compiler

Block Diagram Representation

The internal representation of an HP VEE program is a directed graph constructed of objects that represent devices, with edges between these objects corresponding to the connections visible in the pictorial view. Information about the pictorial presentation is maintained within the internal device graph, but the compiler is only concerned with the connection structure of an HP VEE program.

Transformations

Before the main compilation analysis takes place, the compiler may modify the internal device graph to simplify analysis. These modifications replace constructs having special behaviors by collections of simpler constructs that all have standard behaviors. Such graph modifications only affect the internal representation, and are invisible to the user.

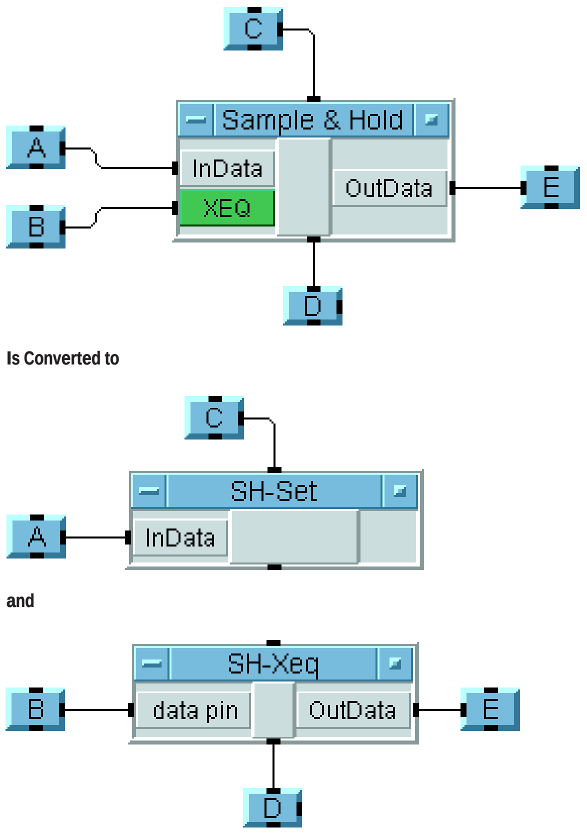
Execute and Control Pin Splitting. For purposes of compilation it is convenient to assume that all nonjunction HP VEE devices fire only after all their inputs receive data. However, some HP VEE devices may activate when only a subset of their inputs have data, typically because such devices have some execute or control pins (described in the section “Pins and Devices” on page 99). To avoid having to consider the types of input pins when performing later stages of compilation, devices with such pins are split into multiple “synthetic” devices such that each synthetic device fires after all its inputs receive data. Synthetic devices are device types that only exist when created by the compiler. They are not part of the user-level HP VEE devices and they never appear on the display.

For example, consider a Sample&Hold device. A Sample&Hold has a data input pin and an execute pin. Data entering on the data pin is copied to a buffer in the Sample&Hold device, but the data is not propagated to the output pin until the execute pin is fired (that is, receives data). **Figure 11** shows how a Sample&Hold device is split into two synthetic devices: SH-Set and SH-Xeq.

The semantic role of SH-Set is to store the data from the data input of Sample&Hold into the buffer associated with Sample&Hold, and SH-Xeq’s job is to put the data in the buffer onto the output pin. Links are made between these devices so the compiler can find either one from the other. The semantic behavior of this collection of synthetic devices, as so connected, is equivalent to the original single device in its context. The HP VEE user interface has the single Sample&Hold device instead of the two separate devices because the tight functional coupling of the two behaviors makes it easier to conceptualize the composite functionality as the action of a single device, making HP

Figure 11

Expansion of execute pins.



VEE easier to use. The compiler does the conversion to simplify compilation.

Like execute pins, control pins also lead to device splitting, but they have a simpler reconnection scheme. Control pins affect the state of a device, but they are not directly involved in determining when a device can fire. They only cause side effects within the device. If a device has N control pins, it is split into N + 1 synthetic devices such that one synthetic device is similar to the original device but with the control pins removed. Each of the other N synthetic devices is associated with one control pin by having the synthetic device's single "normal" data input get the input that had gone to the associated control pin. See **Figure 12** for an example (lines attached to control inputs are drawn with dashed lines in HP VEE). These synthetic control pin devices implement the side effect that the associated control pin was meant to perform. Links are made between the synthetic and original devices to facilitate generating code.

There is a subtlety about control pin semantics that is not addressed merely by splitting. The semantics of control pins dictate that the action they implement take place more or less at the time the pin receives data. Thus if a synthetic device implementing a control pin action has received data, it should be scheduled to execute as soon as possible, ahead of other devices that may also be ready to run. To implement this behavior, control pin devices are tagged as high-priority devices, which cause the scheduler to schedule them for execution before normal-priority devices. This is a general mechanism that can be used for other devices that need to be run at high priority.

A device with both control and execute pins can be expanded by combining expansion techniques in a straightforward way.

Synthesized Constructs. It is sometimes beneficial to split devices that do not have execute or control pins. This is useful for devices having complex semantics that can be implemented with combinations of simpler devices. This can be thought of as using the device level of HP VEE to implement parts of the compiler. It can also be viewed as macro expansion.

The compiler uses this idea to implement the OnCycle device. An OnCycle is split into two synthetic devices and one standard RepeatUntilBreak iterator. The idea is that OnCycle is like a RepeatUntilBreak iterator except it only fires at certain time intervals. This is implemented by initializing a time variable in a synthetic initialization device, then running a RepeatUntilBreak iterator whose output goes into a synthetic device that waits until the proper time, then that waiting device connects to what the OnCycle connected to (see **Figure 13**).

Figure 12

Expansion of control pins.

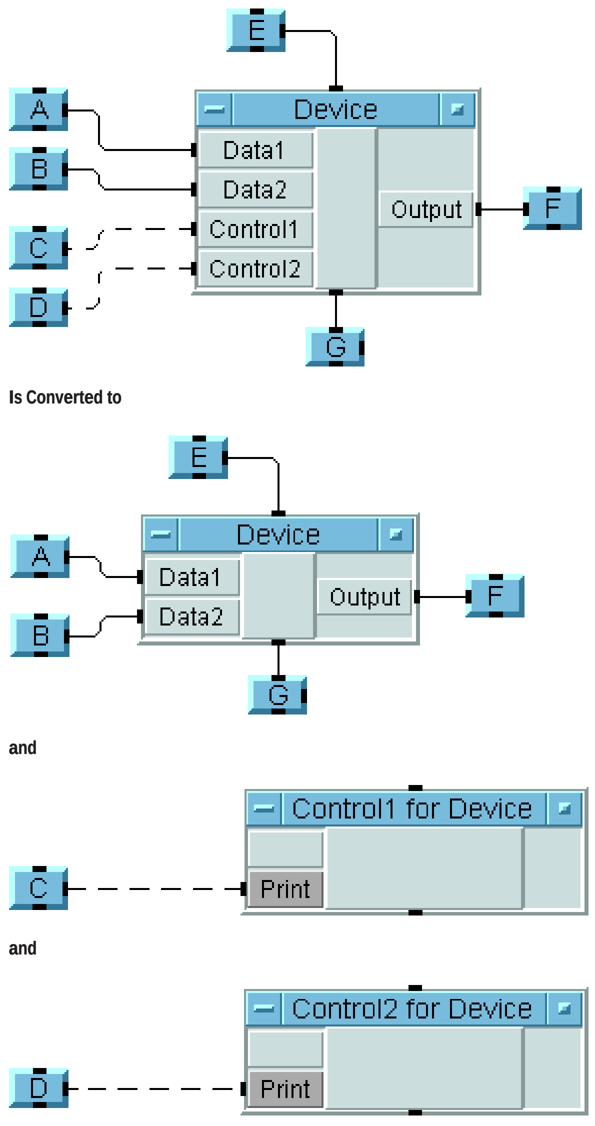
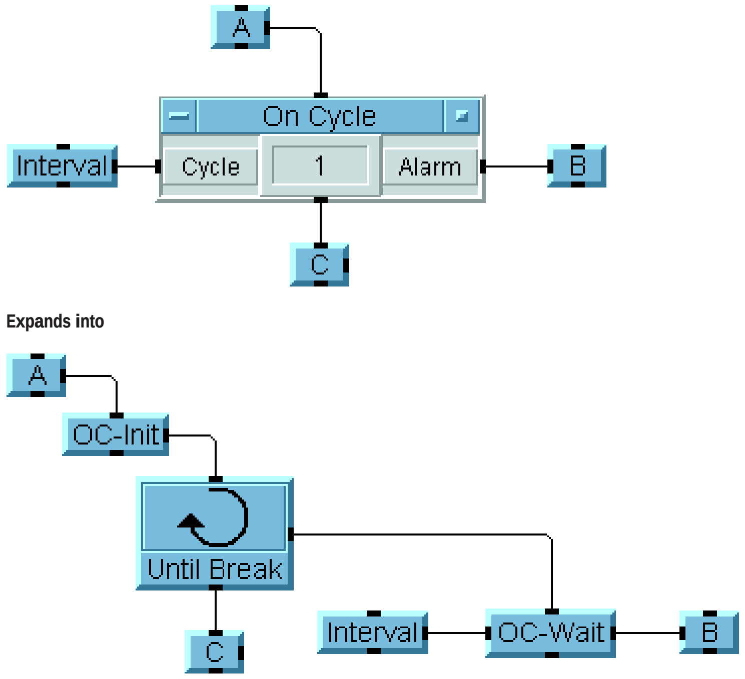


Figure 13

Expansion of an OnCycle *device.*



Scheduling

The scheduler is the part of the HP VEE compiler that determines the order of execution of devices in an HP VEE program. From the HP VEE device graph, it produces a schedule, which is a tree representation of a fairly conventional control-flow program (such as what might correspond to a C program).

Although in general the scheduler determines the program structure, it does not attempt to express the run-time path-branching aspects of special HP VEE constructs. For example, the scheduler treats the If/Then/Else device as a normal primitive device, and therefore assumes all its output pins will fire each time it is executed, although it actually only fires a single pin. The scheduler does this because If/Then/Else can be used in ways that do not directly map into a typical program structure. A later pass of the compiler, called guarding, extends the schedule by adding constructs that represent the flow branching which was ignored by the scheduler. Guarding also takes care of some other situations in which run-time decisions must be made. Guarding is described on page 113.

The basic method used by the scheduler is to traverse the device graph in proper execution order, producing a structure that represents the order and program structure discovered during the traversal. The traversal does not evaluate the program, but only considers basic aspects of the device types. For example, iterators are not traversed multiple times, but the descendants that would be repeatedly executed at run time are determined.

When the scheduler encounters a device, the device is categorized as being in one of four categories, depending on its type. Other than this categorization, the type of the device is ignored by the scheduler. The device categories are: iterators, junctions, asynchronous devices (for example, Delay and Confirm), and everything else. Devices in the last category are referred to as primitive devices. Compilation details for asynchronous devices are not included in this paper. Although the HP VEE language does not have pure data flow semantics, data flow is used as a basic semantic building block. Simple data flow graphs can be serialized using topological sort, often called *topsort*, which is a well known, efficient algorithm.[3] The scheduler is based on topsort, but has significant modifications.

Figure 14

Topsort.

```
Topsort (Graph) {
    Ready–Nodes <– nodes in Graph that have all their input pins fired
    if (Ready–Nodes is Empty)
       return Empty
    else
       Fire outputs of all nodes in Ready–Nodes
       return append(Ready–Nodes, Topsort(Graph – Ready–Nodes))
}

Note:  A node with no inputs is "ready."
```

Topsort takes a directed graph as argument and returns a list of nodes. Each node in the returned list satisfies the criterion that its ancestors occur before it in the returned list. All nodes from the graph that can satisfy this criterion are included. Only nodes that are part of cycles in the graph cannot be topologically sorted. If there is more than one topological sort for a graph, any one is a valid result. See **Figure 14** for a sketch of the topsort algorithm. Here an input is considered to be fired if the output it is directly connected to has been fired. Nodes with no inputs are considered to have all their input pins fired.

For a simple data flow graph, the ancestor/descendant relationship is one of data dependency. The topsort of such a graph lists devices in an order such that a device appears in the list after all the devices that produce data for it. Therefore, in these cases topsort can be used as a device-ordering compilation mechanism, eliminating the need for run-time calculation of what to evaluate next.

Priority Ordering. An HP VEE program that contains only primitive devices and does not use sequence output pins can be scheduled using topsort. However, adding other classes of devices or sequence output pins complicates matters. For this discussion we will consider an HP VEE program to consist of primitive devices, junctions, and iterators. UserObjects will be classified as standard primitive devices. We will temporarily ignore sequence out pins.

The section "Phases of a Sweep" (page 106) described device classes and their prioritized execution order. The scheduler reflects this class-based ordering by extending topsort to keep separate lists of ready devices according to device class, and scheduling items from each class at the proper time.

The structure of the HP VEE program being compiled is reflected in the resulting schedule by having a subschedule computed as a result of the simulated execution of a control (sweep-inducing) device, and storing that subschedule as the *body* of the device. For example, the body of an iterator is that part of the schedule that results from firing the data output pin of the iterator. This results in a hierarchical schedule consisting of a list of devices, with some devices in the list having a subschedule stored as their body. Subschedules are lists of devices, some of which may have their own subschedules.

HP VEE maintains a list of all UserObjects, and their diagrams are compiled one by one at the top level. When they are encountered as a device during scheduling, they are treated like noncompound devices; their contents are not recursively compiled.

The diagram and environment corresponding to the main program or a UserObject subprogram is called a *context*. Internally, a context is represented by a synthetic context device whose substructure includes a list of independent threads (described in the section "Data Flow," on page 101). Each independent thread is a directed graph.

When a context is run, the independent threads run independently and concurrently. The scheduler represents this parallel execution using a Fork abstract syntax constructor, which is also used for parallel iterators and other concurrency. The Fork has a list of threads that run in parallel with each other such that the construct represented by Fork is considered executing while any of its threads are executing. **Figure 15** shows a listing of the scheduler program as described so far.

Figure 15

Basic scheduling.

```
Schedule-Context (Context) {
   for each independent-thread in Independent-Threads (Context)
      // Initialize P, J, and I
      Devices-With-No-Inputs(Independent-thread, &P, &J, &I)
      body(independent-thread) <- Schedule(P, J, I)

   // The body of a Context is a Fork of its independent threads.
   body(Context) <- Make-Fork(Independent-Threads(Context))
}

Schedule (P, J, I) {
   if (P not Empty)
      d <- pop(P)   // Removes first element from P
      Fire-Data-Out-Pins(d, &P, &J, &I)  // May add to P, J, and I.
      return push(d, Schedule(P, J, I))
   else if (J not Empty)
      for each j in J
         Fire-Data-Out-Pins(j, &jP, &jJ, &jI)
         body (j) <- Schedule (jP, jJ, jI)
      return append(J, Schedule(Empty, Empty, I))
   else if (I not Empty)
      for each i in I
         Fire-Data-Out-Pins(i, &iP, &iJ, &iI)
         body(i) <- Schedule(iP, iJ, iI)
      return push(Make-Fork(I), Schedule(Empty, Empty, Empty))
   else
      return Empty
}

Fire-Data-Out-Pins (D, *P &J, *I) {
   For each data output pin, OutPin, of device D, call
   Fire-Pin(Outpin, &P, &J, &I). Return value is not specified.
}

Fire-Pin (OutPin, *P, *J, *I) {
   Fire output pin OutPin. For each input pin IP that
   OutPin is directly connected to, mark IP as ”fired”
   (i.e., as having active data). If IP is attached to device
   D, and firing IP causes D to have its data dependencies
   satisfied, then add D to (the de-reference of) one of P,
   J, or I, which are pointers to variables holding primitive
   devices, junctions, and iterators, respectively. Return
   value is not specified.
}

Make-Fork (Threads) {
   Takes a list of ”threads,” where each thread is a list, and
   returns an object representing a Fork construct where all the
   threads run concurrently with each other. If the list of
   threads is empty, the resulting Fork represents an operation
   that does nothing and returns immediately.
}

Note:  &variable denotes the address of variable, as in C. Within a
formal parameter list, *variable indicates that variable is passed by
reference, analogous to C.
```

The lists P, J, and I in the scheduler of **Figure 15** are lists of devices with satisfied data dependencies that are waiting to be scheduled. These lists hold primitive devices, junctions, and iterators, respectively.

The routine Fire-Pin adds devices to these lists as the devices become ready. When adding a device, Fire-Pin could place the device at the beginning, end, or somewhere else in a list. Placing devices at the beginning leads to a more depth-first traversal, while placing devices at the end is more breadth-first. The semantics of HP VEE do not constrain this aspect of the ordering. The scheduler uses the depth-first option because that ordering can lead to improved efficiency of guard evaluation at run time (see "Guarding Phase Passes" on page 113).

Care must be used when Fire-Pin places junctions on their ready list. Junctions are the only devices the scheduler sees that have their data dependencies satisfied by any subset of their input pins (other cases are removed as described in "Tranformations" on page 107). Thus, when any input pin of a junction is fired, it can be placed on its ready list. However, if a junction pin fires while the junction is currently in the ready list, the junction should not be placed on the list again. This is easy to accomplish by maintaining an *ignore* marker in a junction that is set when the junction is placed on the ready list and cleared after a junction is scheduled. Depending on how junctions are compiled, it may be necessary to record which pins are fired, even when the junction has its ignore marker set.

Sequence Out Pins. HP VEE 3.2 takes a conservative approach by not firing sequence output pins until after executing all devices that can execute without firing a sequence output pin directly in the current sweep (but they can fire in subsweeps). Then a sequence output pin is fired (see "Sequence Pins" on page 102 for more about sequence output pins). A listing of the extended Schedule routine to implement sequence output pins is shown in **Figure 16**.

Figure 16

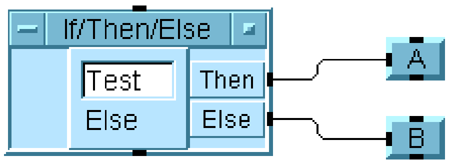
Implementation of sequence out pins.

```
Schedule (P, J, I, S) {
   if (P not Empty)
      d <– pop(P)  // Removes first element from P.
      if (has–sequence–out–pin(d))
         S <– push(seq–out–pin(d), S)   // Add to top of S.
      Fire–Data–Out–Pins(d, &P, &J, &I)  // May add to P, J, and I.
      return push(d, Schedule(P, J, I, S))
   else if (J not Empty)
      for each j in J
         Fire–Data–Out–Pins(j, &jP, &jJ, &jI)
         body(j) <– Schedule(jP, jJ, jI, Empty)
   else if (I not Empty)
      for each i in I
         if (has–sequence–out–pin(i))
            S <– push(seq–out–pin(i), S) // Add to top of S.
         Fire–Data–Out–Pins(i, &iP, &iJ, &iI)
         body(i) <– Schedule(iP, iJ, iI, Empty)
      return push(Make–Fork(I) Schedule(Empty, Empty, Empty, S))
   else if (S not Empty)
      Fire–Pin(first(S), &P, &J, &I)  // May add to P, J, and I.
      return Schedule(P, J, I, rest(S))
   else
      return Empty
}

Note:  Call this, from Schedule–Context, as Schedule(P, J, I, Empty).
```

Figure 17

Simple conditional.



Guarding

As discussed previously, the scheduler ignores the special nature of If/Then/Else and some other constructs. Instead, the guarding compilation phase handles these constructs. The reasons for this division are reviewed here, and the implementation of guarding is outlined.

At first glance it appears that the scheduler could treat each output branch of an If/Then/Else as separate evaluation paths and store the subschedules that start with each branch as separate "bodies" in the scheduled If/Then/Else. For example, the scheduled If/Then/Else in the HP VEE program in **Figure 17** could have a Then body containing device A, and an Else branch containing device B. This could then be compiled into code with a structure like:

```
    if (Test) then A else B.
```

However, conditionals in HP VEE programs can be used in very general configurations that make it difficult to structure the resulting program into a typical If/Then/Else conditional multibranch structure. Descendants of conditionals can overlap in complex ways and can overlap with bodies of junctions and iterators as well. For example, the program in **Figure 18** cannot be structured in a simple nested fashion. The scheduler avoids such issues by treating If/Then/Else like a standard primitive device. Guarding extends the schedule to reflect the run-time branch decisions ignored by the scheduler, but uses a different computational approach that is specialized for this task.

Guarding is in fact used as a general run-time control mechanism, and some generated guards are not associated with HP VEE's If/Then/Else conditionals at all. Some of these will be described below. The various types of conditions that may apply to a device are combined when testing at run-time.

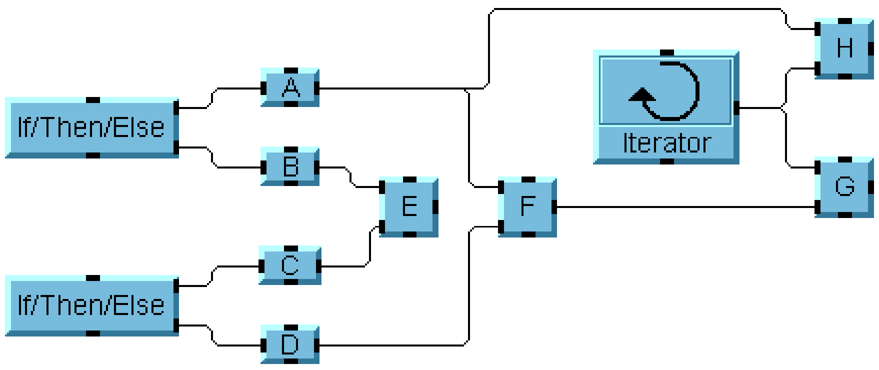
Guarding Phase Passes. The compiler's guarding phase is divided into two passes: guard assignment and guard coalescing. Guard assignment annotates each device in the schedule with a set of guards. This set captures the conditions that must be satisfied at run time for the device to be evaluated. If a device D has assigned to it a set of guards G, then code could be generated for D that reflects this structure:

```
    if G then D
```

Guard assignment annotates devices in the schedule, but the schedule is not otherwise altered. The guard coalescing pass explicitly extends the schedule by inserting synthetic *guarded-body* devices into the schedule, based on the guard annotations made by the guard assignment pass. A guarded-body device has a set of guards assigned to its *guards* property and a subschedule assigned to its *body* property. The guards apply to all elements of the

Figure 18

Overlapping conditionals.



body. After coalescing, guarding is explicitly represented in the schedule by the guarded-body devices and code generation only generates run-time conditions where a guarded-body occurs.

Although correct programs would result if coalescing added a guarded-body to each guarded device individually, the reason for a separate coalescing pass is optimization. Coalescing combines guard sets from adjacent devices into a single guarded-body whose body is a multi-element segment of the schedule. This avoids many redundant run-time conditionals.

To illustrate the redundancy removed by guard coalescing, consider two adjacent devices D1 and D2, both guarded by the same set of guards G. Without coalescing

```
if G then D1
if G then D2
```

is generated. With coalescing,

```
if G then (D1; D2)
```

is generated. More elaborate coalescing can be performed. For example, if D1 is guarded by G, and D2 is guarded by $G \cup H$, then

```
if G then (D1; if H then D2)
```

can result. This last example would be represented in the schedule with nested guarded-body devices of the form:

```
Guarded-Body[G, (D1, Guarded-Body[H, (D2)])].
```

As mentioned previously, when possible the HP VEE program graph is scheduled in a relatively depth-first order to improve efficiency of run-time guard evaluation. The reasoning is that the set of guards for a device is usually a superset of the guards of each of its parents, so scheduling devices adjacent to their ancestors increases the likelihood that adjacent devices have common guards. This works well for coalescing.

Guard Assignment for **If/Then/Else**. Guard assignment for If/Then/Else constructs operate on the list of devices produced by the scheduler. Recall that devices in the schedule occur in an order such that when considering a particular device in the schedule, all devices that produce data consumed by that device will occur earlier in the schedule (except if there is feedback). So a dependency-order traversal of the HP VEE program graph can be accomplished by a simple walk down the schedule list. When we get to a device, we know that its parents have already been processed.

As described in the section “If/Then/Else” on page 103, an If/Then/Else fires a single output pin. Guarding implements the branching implied by this single-pin firing behavior by associating a unique boolean variable, called a *guard*, with each output pin of the If/Then/Else. Before the expressions of an If/Then/Else are evaluated, the guards for all the pins are set to False. When it is determined which pin fires, only the guard associated with that pin is set to True.

Note that since an If/Then/Else can have any number of conditions (and hence, output pins), a single boolean variable for an If/Then/Else is not adequate. A single multi-valued flag could be used instead, but then instead of directly testing its value (True or False), a test would have to use the value resulting from a comparison of the variable with an appropriate value. We will assume here that multiple boolean variables are used.

The basic idea of the guard assignment algorithm is to traverse the schedule list visiting each device, in order, and for each such device D, consider all the direct parents of the device. The parent devices will already have been processed because of the schedule ordering. The guards from the guards set of each of D’s direct parents are added to the guards set of D. Also, if a direct parent of D is an If/Then/Else device, the guard associated with each If/Then/Else output pin that is directly connected to D is also added to the guards set of D. **Figure 19** gives a more detailed sketch of the algorithm.

Other Guarding Considerations. A number of important points have been omitted in the presentation of the guard assignment algorithm. First, it does not address the fact that the schedule is not actually a flat list, but instead is a hierarchical structure because schedule segments are stored as the bodies of devices such as junctions and iterators. One must consider the relationship between the guards on a hierarchical device and the guarding of the body of the device. Also, the interpretation of the guards set must be clarified. These points are related.

First, consider the meaning of the guards sets. These are sets of individual guard objects, where each guard represents a Boolean valued variable. For HP VEE programs without junctions, a guards set can be interpreted as a conjunction (logical AND) of the individual guards in the set. This is because after the graph transformations performed earlier in the compilation (see section “Transformations,” page 107), all nonjunctions the compiler sees must have data on all their input pins to run. The guards

Figure 19

Basic guard assignment for If/Then/Else.

```
// These do not have specified return values.

Guard–Assignment (Schedule) {          // Schedule is a list of devices.
   if (Schedule not Empty)
      Assign–Guards–to–Device(first(Schedule))
      Guard–Assignment(rest(Schedule))
}

Assign–Guards–to–Device (D) {           // D is a device.
   for each input pin IP of D
      OP <– output–pin–connected–to(IP)
      Parent <– device–attached–to(OP)
      guards(D) <– guards(D) ∪ guards(Parent)
      if (Parent is an IF/Then/Else device)
         guards(D) <– guards(D) ∪ {guard(OP)}
```

inherited from parent devices represent the conditions needed for each parent to run, and also for direct If/Then/Else parents to place data on the appropriate lines. Thus, having data on all input lines requires all the guards to be true.

A problem arises when HP VEE programs contain junctions. Junctions can run when any subset of their inputs have data, so the guards on junctions must be disjunctive (logical OR) instead of conjunctive. If we allow both conjunctive and disjunctive guarding, the guard sets must be replaced with potentially complex boolean expressions (using conjunctions and disjunctions). We show how to solve this problem below.

Guards assigned to an iterator device should not be propagated into the iterator body. If the guards on the iterator device are false, the iterator will be skipped, so the body will be skipped as well. If the guards are true, the iterator and its body will run. In general, the body implicitly inherits the effect of the guards on the iterator. This works for junction bodies as well, so that the disjunctive guarding implied by junction semantics does not need to produce any explicit guards in the junction body. We can conclude from what we have discussed so far that guards do not need to propagate into the bodies of junctions or iterators, and the guard set can indeed be considered to represent a conjunction.

However, not propagating the guards of junctions and iterators into their bodies can lead to a problem when data created inside the body is used outside the body. We will refer to such data as “escaping” from the junction or iterator. Consider the configuration in **Figure 20**. Here device B is in the iterator body, but device C is not. If the If/Then/Else evaluates such that the condition connected to the iterator is false, the iterator and its body (device B) will

Figure 20

Data “escaping” from an iterator body.

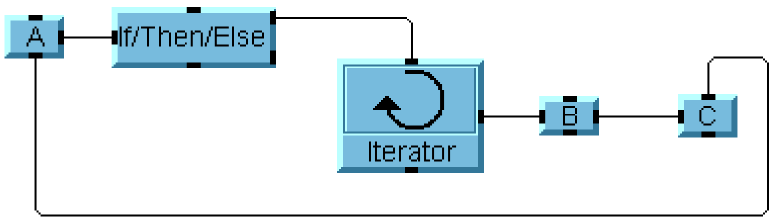
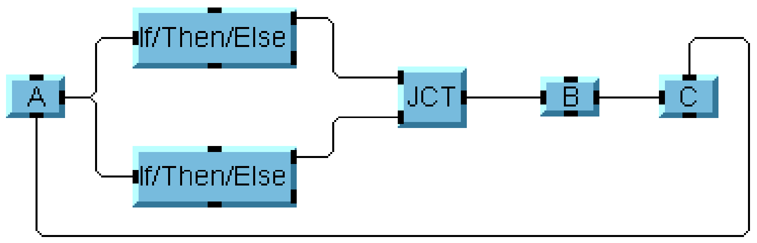


Figure 21

Data escaping from an junction body.



be skipped because the iterator is guarded via the If/Then/Else. If the iterator does not run, device C should not run either, because it will not have input data. However, if the body of the iterator does not explicitly have the iterator's guards, device C will not inherit any guards from its parents. Here device C should be guarded by the guard that is on the iterator device. **Figure 21** shows a similar configuration using a junction instead of an iterator. Here device B does not need an explicit guard because it is in the body of the junction, but device C needs a disjunctive guard.

Although this problem could be solved by having guards of junctions and iterators propagate along paths of escaping data, this is not the best solution. One problem would be that disjunctive guards would still be needed. Another problem is that such propagated guards are sometimes redundant, and often more complicated than needed. A better and more general solution is motivated by the example in **Figure 22**. Here data escapes from the iterator, but there are no explicit conditionals. The iterator iterates the number of times specified by the formula feeding into it.

For the type of situation illustrated in **Figure 22**, it cannot, in general, be determined at compile time how many times the iterator will iterate. If it iterates zero times, device B will be skipped because it is in the iterator body, but device C should not run either. Thus, it needs to be determined how device C should be guarded. There is no conditional to generate a guard. To solve this problem, a new type of guard was created, called a *ran-once* guard. A ran-once guard is associated with the iterator, and is initially set to False. In some cases it may need to be reset to False in outer sweeps (discussed below). If the iterator runs, the ran-once guard is set to True. Devices that use data escaping from the iterator, such as device C in the example, should be guarded by the iterator's ran-once guard.

So we see that ran-once guards are, in general, needed for iterators. Devices receiving escaping data should inherit not just the guards from the guards set of the iterator device, but also from the ran-once guard for the iterator, conjoined together (that is, they all need to be true). The key observation is that this conjunction has a value that is identical to the ran-once guard alone. The ran-once guard encapsulates the guards on the iterator because the ran-once guard indicates whether or not the iterator runs, no matter what the reason. It might not run because the iterator count is zero, but it also might not run because of guarding the iterator device. Both reasons are captured by the ran-once guard. Thus, it is sufficient to guard devices that use escaping data with just the ran-once guard.

Extending this idea to junctions, a ran-once guard for junctions encapsulates the disjoin of the guards inherited by the junction. Therefore, devices that use data that escapes from a junction body can be guarded by the junction's ran-once guard, and no explicit disjoined guards are needed.

Figure 22

Device C should not run if the iterator iterates zero times.

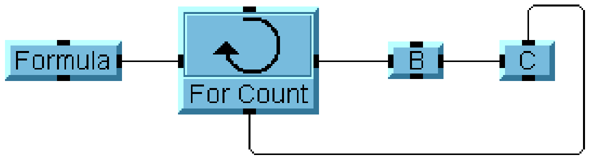
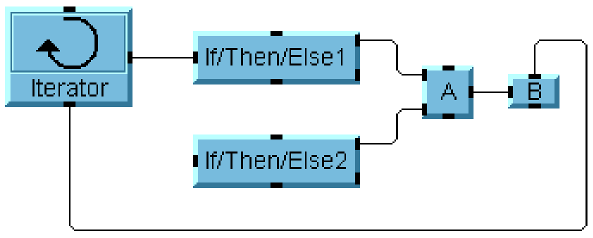


Figure 23

Conditionals flowing into and out of a conditional body.



It can now be seen that the complexity of guard sets is kept relatively low by not propagating guards through junctions and iterators to their bodies and by only propagating ran-once guards for data that escapes a body.

It should be noted that guards created inside the body of a junction or iterator can propagate outside the junction or iterator via escaping data, and guards created outside the body can propagate into the body if they do not enter through the body's junction or iterator device. For example, in **Figure 23**, If/Then/Else1 and device A are both in the body of the iterator, and If/Then/Else2 and device B are outside. The guard from If/Then/Else2 propagates into the iterator body to device A, and the guard from If/Then/Else1 propagates out of the body to device B (as does the guard from If/Then/Else2). Devices A and B are both guarded by If/Then/Else1 and If/Then/Else2.

At run time, If/Then/Else guards do not have to be initialized before reaching their If/Then/Else device. This is because if the If/Then/Else is not reached, it will be because the If/Then/Else or containing devices are suppressed by other guards (possibly ran-once guards encapsulating some of them), and these surrounding guards propagate along with the If/Then/Else guards. So if the If/Then/Else is suppressed by guards, these guards will also suppress descendants of the If/Then/Else that would have used the If/Then/Else guard (because the guards are conjoined).

Ran-once guards, however, must be initialized to False at the start of sweeps that use them because if the junction or iterator controlling a ran-once guard is suppressed by other guards, those surrounding guards are not propagated along with the ran-once guard. This initialization maintains the active data rule semantics described in the section “Iterators” (page 103), where data created in a sweep is invalidated if the sweep restarts. If this initialization were not done, a ran-once guard could remain set to True from an earlier sweep, allowing invalidated data to be used. For example, in **Figure 24**, Iterator2 will run in the first iteration of Iterator1 (when it outputs 1), but it will not run in the second iteration of Iterator1 (when it outputs 2). If the ran-once guard for Iterator2 were not reset at the start of the second iteration of Iterator1, it would still be True from the first iteration, and device A would evaluate in the second iteration of Iterator1, but it should not. In general, a ran-once guard needs to be initialized in the innermost sweep (junction or iterator) containing the junction or iterator it is associated with, and in all sweeps containing that sweep up to and including the sweep that also contains all of the consumers of the ran-once guard (which may be the top level).

An additional point concerning ran-once guards is that when data escapes from nested iterators or junctions (or both), only the ran-once guard of the deepest one needs to be used. Consider the program in **Figure 25**. The structure of the schedule for this example is:

- Top-level schedule: Iterator1
- body(Iterator1): Iterator2

Figure 24

The ran-once guard at Iterator2 *needs to be set to* False *at the start of each iteration of* Iterator1*, or device A would run too often.*

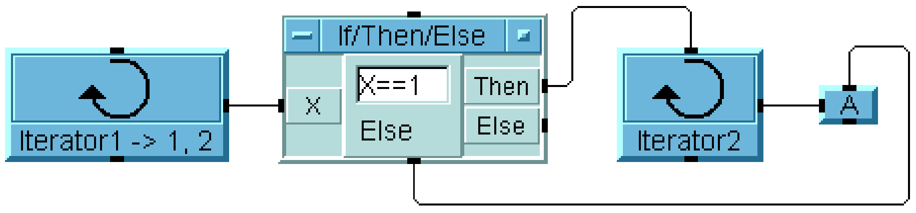
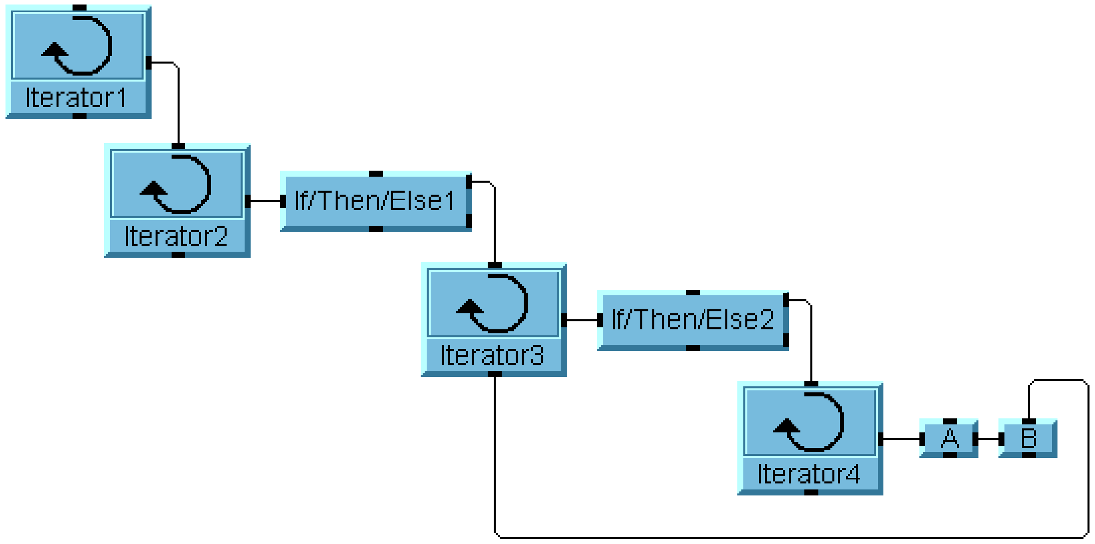


Figure 25

Data escaping from nested iterators.



- body(Iterator2): If/Then/Else1, Iterator3, B
- body(Iterator3): If/Then/Else2, Iterator4
- body(Iterator4): A

Since B uses data created in Iterator4, B needs to be guarded by the ran-once guard for Iterator4. It does not, however, need to also be guarded by the ran-once guard of Iterator3, even though the data used by B is escaping from both Iterator4 and Iterator3. The ran-once guard for Iterator3 is not needed at all in this example. The ran-once guard for Iterator4 must be initialized to False at the start of each iteration of Iterator2 and Iterator3. It is set to True at the start of the first iteration of Iterator4. It is not necessary to set it at the start of each iteration of Iterator4, other than the first iteration, since its value cannot change during Iterator4's execution. The ran-once guard does not need to be initialized by Iterator1 or at the top level, because it is not used at those levels. Also, a guard from If/Then/Else1 will guard Iterator3 and B, and a guard from If/Then/Else2 will guard Iterator4.

An extended version of Assign-Guards-to-Device from **Figure 19** appears in **Figure 26**. This version implements the methods discussed above.

Guards are used for control of other constructs as well. One example is error pins. An error output pin can be placed on many HP VEE devices. If present, errors in the device are trapped by the system and cause the error pin to fire instead of the other output pins. If there is no error the other pins fire and their error pin does not. From the compiler's point of view, any device with an error pin is similar to a two-branch If/Then/Else, except that the nonerror branch can be associated with any number of output pins. Thus, guards are used to implement error outputs in a similar manner to the implementation of If/Then/Else. The guards are set by run-time error trapping constructs compiled with the implementation of the device with the error pin.

Sometimes guards can be optimized away at compile time. Ran-once guards are only needed for escaping data, so if no data escapes they do not have to be set. If data does escape, but it can be verified at compile time that the innermost escaped sweep runs sufficiently often compared to all the devices that consume the escaped data, the ran-once guard can be omitted. For example, if an iterator is known to run a fixed nonzero number of times, and there are no guards acting on it (directly or indirectly), then it does not need to have a ran-once guard for any escaping data because it will verifiably run at least once. There are many other situations where various types of guards can be optimized away, although in practice it can be complex.

Figure 26

Guard assignment.

```
Assign–Guards–to–Device (D) {  // D is a device.
   if (D is a junction or iterator)
      // Assign guards to body before D itself so that body won't
      // inherit guards from D.
      Guard–Assignment(body(D))
   for (each input pin IP of D)
      OP <– output–pin–connected–to(OP)
      Parent <– device–attached–to(OP)
      guards(D) <– guards(D) ∪  guards(Parent)
      if (Parent is an If/Then/Else device)
         guards(D) <– guards(D) ∪  {guards(OP)}
      if (Parent is in a junction or iterator body that D is not in)
         RanOnce–Guard <–
           ran–once–guard(deepest–junction–or–iterator–in(Parent))
         guards(D) <– guards(D) ∪  {RanOnce–Guard}
         record <Parent,D,RanOnce–Guard> for RanOnce initialization
}
```

Type Annotation

Type annotation is a phase of the prototype HP VEE compiler in which every pin of every device in an HP VEE program is annotated with a description of the types of data it will handle at run time. The type annotations can often be used to generate more efficient run-time code. Some run-time checks can be eliminated and composite data objects can be replaced by hardware supported representations (for example, real numbers). The type descriptions are conservative approximations in that they err on the safe side. For example, we may obtain a conservative approximation that some data input pin is always a real or integer scalar, when in fact it could only be a real scalar.

Recall from the discussion of HP VEE data types that data objects in HP VEE are typed, but pins and connections are not typed. In HP VEE 3.2 the data objects have fields that specify the type information. Also, recall that most HP VEE devices will handle many different types of data as input. Every time a device is executed by the HP VEE 3.2 interpreter, the following steps typically occur:

1. The type of each input value is ascertained and compared against the pin's required type. A conversion may need to be performed, possibly allocating and initializing a new data object.
2. A check is made to determine whether input values need to be promoted to a common type. Again, input values may need to be converted.
3. The data inputs, or their conversions, are used to perform the computation.
4. The final result is injected into a data object, along with type information.

When executing a simple numerical device, these sorts of extraction, conversion, and injection operations on data objects can account for a substantial percentage of the device's execution time. With type annotation, unnecessary extraction, conversion and injection operations can often be eliminated. The result can be very efficient numerical code that uses standard hardware-supported data representations such as real and integer.

The type descriptions generated during type annotation are sets of ordered pairs. Each pair specifies a class of values and a class of shapes, as shown in **Table I**. A type description represents a disjunction of pairs, as shown in **Table II**. Type descriptions are normalized so that the number of pairs is minimized. Normalizing a type description does not change its meaning, as shown in **Table III**.

Table I

Type Descriptions

Pair	Meaning
<real, scalar>	Data is a real scalar
<real, array>	Data is a real array of some dimension
<real, array-1d>	Data is a real one-dimensional array
<real, any>	Data is a real array or real scalar
<real, any>	Data in anything
<any, array>	Data is an array, but type is unknown

Table II

Disjunction of Pairs in Type Descriptions

Type Description	Meaning
{<real, scalar>, <integer, scalar>}	Data is a real or integer scalar
{<real, scalar>}	Data is a real scalar
{<real, scalar>, <nil, any>}	Data is a real scalar or nil

Table III

Normalization of Type Descriptions

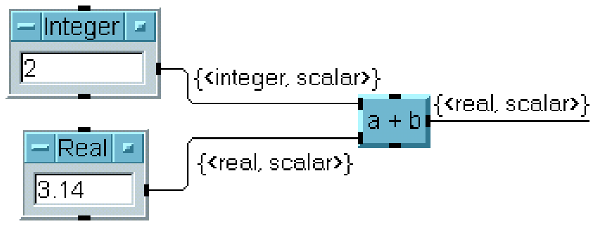
Type Description	Normalized Type Description
{<real, scalar>, <real, any>}	{<real, any>}
{<real, scalar>, <real, array>}	{<real, any>}
{<real, scalar>, <integer, any>}	{<real, scalar>, <integer, any>}

The type descriptions are ordered by a < b if the set of values described by a is a proper subset of the values described by b. The bottom element of the ordering is {} and the top element of the ordering is {<any, any>}.

Determining the type description for every pin of every device in an HP VEE program can be viewed as executing the HP VEE program over the abstract domain of type descriptions rather than over the standard domain of values. In this view, devices take type descriptions as input and compute appropriate type descriptions as outputs. The resulting type descriptions are propagated to descendants for further annotation. This notion of computing with abstract values is sometimes called *abstract interpretation*. An annotated program is shown in **Figure 27**. In this figure, the addition device takes {<integer, scalar>} and {<real, scalar>} and produces {<real, scalar>} as the result. The reason for this is that the standard addition device promotes its inputs to a common type before computing the result, and the abstract addition device reflects this.

Figure 27

Example of type annotation.



Since truth values cannot be determined from type descriptions, when performing type annotation we simply ignore all guard and control values and propagate type descriptions to all output pins.

Type annotation is performed by traversing the scheduler output and determining type annotations for every device that is encountered. Because of feedback loops, type annotations can change on input pins of devices that have already been visited in the traversal. Thus, the type annotator may need to traverse the scheduler output multiple times until all type annotations have stabilized. The subsequent traversals are very efficient because only devices that need to have their output pin annotations updated are redone. Type annotations do eventually stabilize, because when a type description, d, is changed, it is replaced by a type description that is strictly greater than d in the ordering. Since there are only a finite number of type descriptions, only a finite number of changes are possible.

Code Generation

The internal structure that results from the sequence of graph transformations, scheduling, guarding, and type annotation, is similar in form to an annotated parse tree that might be produced by a conventional compiler. To complete the compilation, target code is produced from this structure. If appropriate libraries of HP VEE built-in routines exist, most of the compiled internal structure can be straightforwardly mapped to any conventional programming language, such as C. The mapping may be difficult for error trapping (for error pins) and the implementation of the Fork construct. Fork should map to a construct (or constructs) to implement thread creation and destruction. Error trapping is built-in or can be implemented in most programming systems, and threads have become reasonably common. If the generated code is interpreted, threading is easy to implement.

To generate code for different target languages, only the code generator needs to be modified. This is a relatively small part of the compiler. Prototype code generators for C, and for an interpreted byte-code, have been implemented. C generators have been written to produce standalone programs for subsets of HP VEE and to produce C code to be compiled and dynamically loaded into a running HP VEE 3.2 session. A byte-code generator was written to produce a stream of byte code to a file or in-memory array. A simple interpreter running inside HP VEE can run the byte code. An advantage of the byte-code generator over the C code generator is that an external compiler is not needed. Another advantage of interpretation is that program development features such as single stepping and tracing are easier to implement, especially in the context of running programs inside the HP VEE development system.

Conclusion

An overview of the major components of a compiler for HP VEE 3.2 has been presented. Although many details were omitted, the fundamental algorithms for a large class of programs have been explained. The five compiler components are graph transformation, device scheduling, guard assignment, type annotation, and code generation. These components combine to implement the control semantics explicitly or implicitly specified in an HP VEE program. These semantics were also described. A compiler based on the prototype compiler has now become an integral part of HP VEE 4.0.

Acknowledgments

A compiler for HP VEE was originally suggested to us by Randy Coverstone, our department manager at Hewlett-Packard Laboratories. Wes Higaki and Bill Shreve also provided valuable management support. Special mention goes to the following Measurement Systems Division engineers from the HP VEE product team: Jim Bachman, Ken Colasuonno, John Dumais, and Susan Wolber. These four had the difficult task of understanding our prototype and making it into a product. The result of their efforts is the compiler in HP VEE 4.0. Bill Hunt and Randy Bailey were engineers on the Measurement Systems Division HP VEE team who worked with us early in the project. Bill helped us integrate early versions of the compiler with HP VEE, and supplied much other help as well. John Bidwell, Bill Heinzman, and Chuck Heller also provided encouragement and support.

References

1. R. Helsel, *Cutting Your Test Development Time with HP VEE: An Iconic Programming Language*, P T R Prentice Hall, Englewood Cliffs, NJ, 1994.

2. *Hewlett-Packard Journal*, Vol. 43, no. 5, October 1992, pp. 72-88.

3. D. E. Knuth, The Art of Computer Programming, Vol. 1/Fundamental Algorithms, Addison-Wesley, 1973.

Online Information

More information about HP VEE can be found at:

http://www.hp.com/go/HPVEE

WWW

▶ Go to Journal Home Page