A Mental Framework for use in Creating Hardware Independent Parallel Languages

by K. Sean Halle
University of California at Santa Cruz

Email: seanhalle@yahoo.com

Web: codetime.sourceforge.net

Writing completed: May 25, 2006

Abstract

This paper presents a non-formal framework which may help in understanding the hardware-independence properties of computation models and the programming languages built upon them. This framework models a language as a *base-case-interface* and a program as a *processor-specification* that is stated in terms of a base-case-interface. Each of the elements of the framework is defined and its interactions are shown.

This paper then presents several things: it applies the framework to Lambda Calculus; it uses the framework to divide programming languages into categories; and it shows how the framework can be used to create new parallel languages that have desirable hardware-independence properties.

1 Introduction

Hardware-independent parallel programming is increasing in importance due to the major shift in silicon-chip processors, which are transitioning from a single thread on a chip to an exponentially increasing number of threads on each successive generation of chip.

We could use a framework that is helpful in guiding the creation of new computation models that support hardware independent parallel programs. This paper suggests such a framework, with the caveat that it is not a formal framework in the sense of being usable to prove theorems. However, we have attempted to make it precise enough to be useful. Our intention in describing the framework is to provide a mental tool which may hopefully be of assistance to language designers for parallel programming languages.

This framework is arbitrary. Despite this, it has its uses. It is also different than the existing frameworks used for many years. A mental framework reaches deep in a person's thinking, therefore having one framework replaced with another can be bewildering. The authors hope that the benefits of this par-

ticular choice of framework outweigh the effort required to understand it.

Section 2 defines each element of the framework, illustrates the inter-relationships among the elements, and gives examples of their use. Section 3 applies the framework to Lambda Calculus, uses the framework to categorize several programming languages, and gives more detail on scheduling constraints. Section 4 states what is meant by hardware-independence, and states what features a language could have that would expose maximum parallelism, enable wide hardware-independence, and enhance programmer productivity. Section 5 wraps up with a listing of terms defined and conclusions drawn in the paper.

2 Description of the Framework

2.1 Framework

The framework distinguishes *specifications* of processors from *active* processors.

2.1.1 Processor-Specifications

A *processor-specification* has an *interface* and *internals*. A processor-specification's internals are patterns stated in terms of the *interface-elements* of other processor-specifications. In turn, the lower-level processor-specs may have *internal-patterns* stated in terms of the interface-elements of even lower-level processor-specs, and so on. The sequence forms a directed graph of processor-specifications, called a *processor-specifcation-graph*, or spec-graph for short.

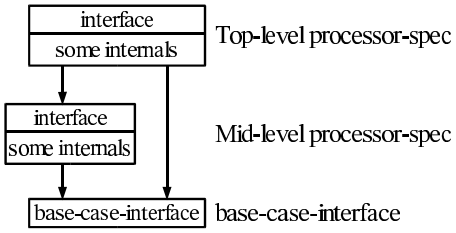

Figure 1. A simple directed graph of processor-specifications. The internals of one processor-spec are stated in terms of the interface-elements of lower processor-specs.

One or more *base-case-interfaces* terminate each spec-graph. A base-case-interface has a *stated-external-behavior* instead of internals. It is treated as a black box; nothing below the interface is specified.

Each processor-specification has exactly one base-case-interface in its *interface-element-name-space*. The processor-spec relies upon special elements of the base-case-interface to provide the infrastructure within which to state the processor-spec's interface, and to provide the "scaffolding" for stating the processor-spec's internal-patterns. The given processor-spec's interface-element-name-space must also contain the interface-elements that come from other processor-specifications and appear in its internal-patterns.

A language specifies a base-case-interface. A language also provides *interface-element-name-space*

commands, which take part in the construction of "concrete" processor-specification-graphs. Appearing in subsection ⟨reference||⟩ are more details on interface-element-name-space commands, how they participate in the construction of a processor-specification's interface-element-name-space, and how they participate in the construction of a processor-specification-graph.

Processor-specifications are organized in *processor-specification-units*. A language defines what a processor-spec-unit is. In C, for example, the processor-specification-unit is the file, which may contain multiple procedure-definitions. Through this framework, a procedure-definition is seen as both defining interface-elements – the procedure name and parameter-pattern – and defining part of one of the internals – the lookup-table – of a processor-spec.

Which procedure-names end up being grouped together in the interface of a single processor-spec is determined differently by each language. It may be arguable for a given language what is the agreed-upon preferred way to determine how to group patterns given in processor-specification-units into individual processor-specs. In C, a good candidate might be that each level of procedure call within the procedures named in a single header file defines a single processor-spec. In C++ and Java, a good candidate might be that each level of method call within a single class defines a single processor-spec.

2.1.2 Active-Processors

Meanwhile, an active-processor is an *animated* processor-specification. An active-processor is made by animating a processor-specation with one or more other active-processors. This forms an *active-processor-graph*.

Active-processor-graphs are cousins of processor-spec-graphs but are not the same. A single processor-spec may be animated several times. Each animation of the same processor-spec is a different active-processor appearing in the active-processor-graph.

An animating active-processor must present an interface that matches the interface linked-to in the interface-element-name-space of the animated processor-spec. Given that, if the animating active-processor's *observed-behavior* matches the *derived-behavior* of the processor-spec whose interface appears

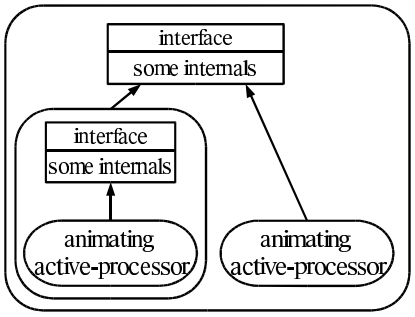

Figure 2. A simple active-processor-graph. Each rounded-corner-box represents an active-processor. The mid-level processor-spec is animated by a single active-processor. The top-level processor-spec requires two active-processors, one for each kind of interface-element appearing in its internal-patterns. Each animated processor-spec is an active-processor in its own right.

in the animated processor-spec's interface-element-name-space then the animated processor-spec will display observed-behavior that matches the derived-behavior of the processor-spec.

Notice that an active-processor presents the interface of a single processor-spec. No means exists to detect any of the other active-processors that might lie beneath that active-processor's interface. Thus, from the "user" point of view, an active-processor is a single indivisible entity with an interface and observed behavior. For example, one cannot detect the difference between a single-issue silicon-chip and an aggressive out-of-order silicon-chip by observing program-results. They both present observed-behavior that conforms to the Instruction Set Architecture interface. This is true even though the out-of-order chip has many more internal active-processors when powered-up. Another example of an un-detectable internal active-processor is an animated "private" method in an animated Java-class.

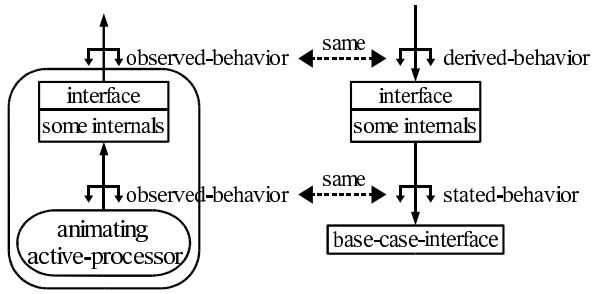

Figure 3. For active-processors, one observes behavior. However, for processor-specs, one either derives behavior or one states behavior. One wishes an animated processor-spec to display observed-behavior that matches the derived-behavior of the processor-spec which is animated. This match will happen if the active-processor which replaces the base-case-interface has observed-behavior that matches the stated-behavior of the base-case-interface.

The derived-behavior of a processor-spec-graph is the set over all valid received-patterns of the result-pattern returned for that received-pattern by the processor-spec-graph. The process of deriving behavior of a processor-spec on a particular received-pattern is the process of an active-processor animating a processor-spec which then receives that received-pattern. The lay term "run a program on an input" is defined precisely in this framework as "derive the behavior of the program's processor-spec-graph when the top-level processor-spec receives a given received-pattern".

Additionally, active-processors have state. The state of a single animated processor-spec is called an *activation*. An activation consists only of the state of a single node in an active-processor-graph. This state consists of "working" state, "bookkeeping" state, and an *interpreter-internal-representation* of the processor-spec. Part of the process of animating a processor-spec is creating some physical form of interpreter-internal-representation from the *pre-interpreted-representation* of the processor-spec. More details on the kinds of activation state and interpreter-internal-representations are in subsection ⟨reference|⟩.

At this point in the paper, an important point can be made about processor-specifications. Not all internals need to be specified in a given processor-spec. However, every "sufficiently powerful" active-processor must contain all internals. The active-processor which animates a processor-spec must supply any missing internals. Typically, the active-processor which presents the base-case-interface provides most of the internals. "Sufficiently powerful" is not defined in this paper, however, the Turing Machine example in subsection ⟨reference|⟩ provides some intuition about the term.

An active-processor that presents a base-case-interface is called an *interpreter-active-processor*. This kind of active-processor is similar to the lay-concept of interpreter. However, an interpreter-active-pro-

cessor does not provide all the behavior associated with what is commonly called an interpreter. There are many distinctions between the precise term "interpreter-active-processor" and the lay-term "interpreter", the most important difference at this point in the paper is that an interpreter-active-processor only provides the behavior of the base-case-interface-elements. It does not provide the behavior of the interface-element-name-space commands. More details on interpreters and compilers is given in subsection ⟨reference|⟩.

2.1.3 Examples and applications of the defined terms

Now we give short examples of many of the defined terms, then apply terms together in larger examples.

- pre-interpreter-representation processor-specification-graphs: programs in source form; circuit-schematics of a chip.

- interpreter-internal-representation processor-specification-graphs: memory-resident executable images of programs; un-powered silicon chips. Notice that the only difference between a program executable in memory and a silicon chip is the medium used to represent the processor-specification-graph – patterns of charge among capacitors vs. patterns of silicon.

- base-case-interfaces: silicon-chip Instruction Set Architecture specifications; (portions of) programming-language specifications; the quantum-mechanics-laws of device-physics (upon which a processor-spec-graph is built that derives the stated-behavior of the base-case-elements in a circuit-schematic of a silicon-chip as shown in detail in figure 4).

- active-processors: a running process; a powered-up silicon chip; a human performing program steps by hand. Each of these may in turn have an internal active-processor-graph, each node of which may be another active-processor-graph, and so on, down to the interpreter-active-processors that correspond to the base-case-interface-elements. Notice that each base-case-interface-element can have its stated-behavior be the derived-behavior of a separate processor-spec-graph, as is done in the example shown in figure 4. The choice of which base-case-interface to stop with is effectively arbitrary. In the example in figure 4, the base-case-interface chosen to be the final, lowest level, base-case-interface is the laws of quantum-mechanics for silicon devices.

- activations (which all include an interpreter-internal-representation of the animated processor-spec): an instance of a class in an object-oriented program (IE, an object on the heap); the internal state of a Java interpreter-active-processor that is stored on the stack during a method-call; the register-state of a silicon-chip interpreter-active-processor that is stored on the stack during execution of a `call` instruction. Notice that the same silicon-chip interpreter-active-processor often animates several processor-specs by saving and restoring the bookkeeping state of each activation in turn; see subsection ⟨reference|⟩ for more about interpreter-active-processors.

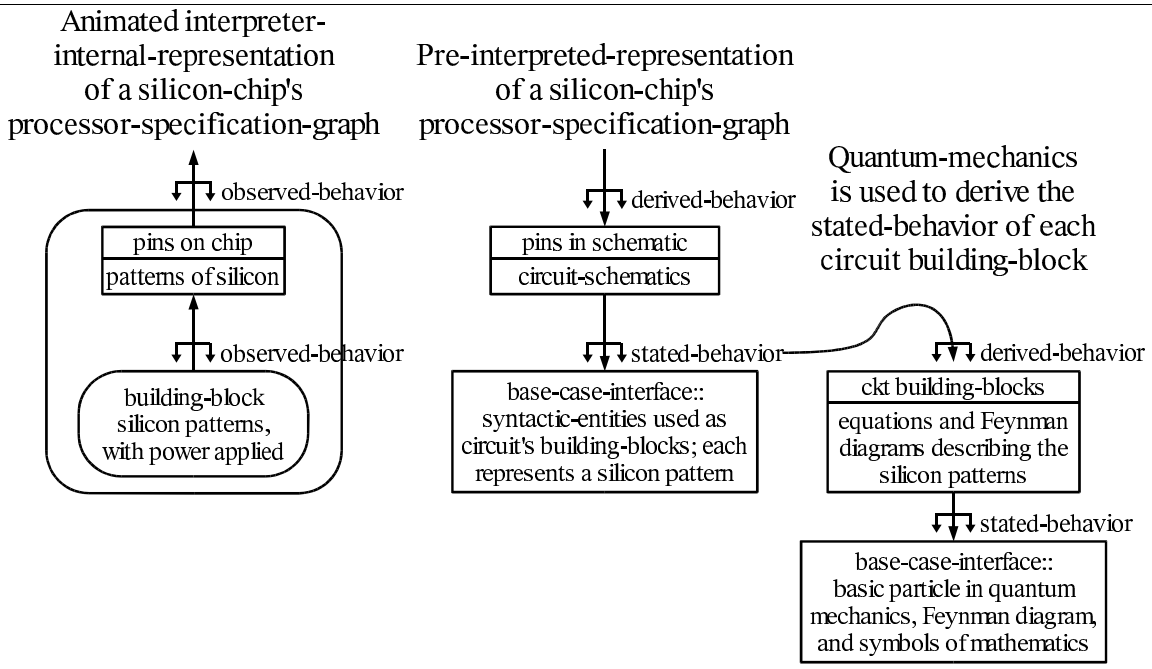

Figure 4. A silicon-processor-chip. The pre-interpreter-representation of the chip's processor-spec-graph is in the middle. This form of the processor-spec-graph has a single level of circuit-schematics (which is simplified from a real chip-design). Each schematic is a pattern stated in terms of syntactic circuit-elements such as transistors and wires. The set of circuit-elements is the base-case-interface (more precisely, the base-case-interface's set of interface-elements). A silicon-compiler (layout engineer or Cadence synthesis tool plus foundry) turns the pre-interpreter-representation into an interpreter-internal-representation, which is a pattern in terms of physical silicon structures. Each circuit-element in the pre-form matches to a physical structure of silicon in the internal-form.

On the far right is another processor-spec-graph. This one has a top-level interface that is the same as the base-case interface of the middle graph. Hence, the right processor-spec-graph has a derived-behavior for each circuit-element. This derived-behavior is used as the stated-behavior of the base-case-interface-elements in the middle-graph.

The right-hand processor-spec-graph states an internal-pattern for each physical silicon structure in terms of, say, Feynman diagrams, plus mathematics symbols. Math syntax, grammar, and stated-behavior plus Feynman diagram syntax, grammar and stated-behavior are the base-case-interface of the right-hand processor-spec-graph.

A powered-up chip is an active-processor whose interface is the electrical signals on the chip-pins. The patterns of silicon created by the foundry are the interpreter-internal-representation of the middle processor-spec-graph. These patterns are animated by a number of active-processors. Each transistor-structure receiving power and each wire-structure receiving an electrical signal acts as a separate active-processor. Each of these active-processors is the physical-form corresponding to a circuit-element. That structure's physical nature then produces observable-behavior.

If the derived-behavior of the right-hand processor-spec matches the observed-behavior of each physical-structure active-processor, then the pins of the powered-up chip will display observed-behavior that matches the derived-behavior of the middle processor-spec. If this derived-behavior in-turn matches the stated-behavior of the chip's specification, then the powered-silicon-chip displays observed-behavior that matches the chip's specification-stated-behavior. Then, the chip's specification can be used as a base-case-interface for program-processor-specs intended to be animated by one of these powered-up chips.

A complex example which involves many defined terms is given in figure 4. This example shows that in order for a powered-up silicon-chip to display observed-behavior that matches the chip's circuit-schematic processor-spec-graph's derived-behavior, the structures of silicon on the powered-up chip that

correspond to circuit-schematic base-case-interface-elements must display observed-behavior that matches the stated-behavior of the circuit-schematic base-case-interface. Quantum-mechanics is used as a base-case-interface of a separate processor-spec-graph. This second processor-spec-graph models a structure of silicon for each circuit-schematic base-case-interface-element. The derived-behavior of this second processor-spec-graph is used as the stated-behavior of the base-case-interface-elements at the bottom of the circuit-schematic processor-spec-graph.

This example shows the difference between mathematics and the real world. Math is a base-case-interface plus a processor-spec-graph built on top of it. Humans act as interpreter-active-processors that replace the base-case-interface. Humans use feedback from one person to another to guarantee that the interpreter-active-processor has exactly the same observed-behavior as the stated-behavior of the base-case-interface. If there is a difference between observed-behavior and stated-behavior, it is the active-processor that must change. In contrast, in the real world if we have circuit-element stated-behavior that is different from observed-behavior of the silicon-structures, then it is the base-case-interface that must change. People care what base-case-interface we choose, and modify their observed-behavior to match stated-behavior. The real world doesn't care what base-case-interface we choose, we must modify our stated-behavior to match the real world observed-behavior.

As a last example of the use of the defined terms, we can state what a bug is. The specification of an application program is a base-case-interface. It states the commands the program accepts; this is the set of base-case-interface-elements. The application-spec also states the behavior expected from each command; this is the stated-behavior of the base-case-interface. Given this, testing program-code is the process of observing the behavior of an active-processor in response to a variety of *input-patterns*. The active-processor observed is the animation of the program's processor-spec-graph. A bug is detected when the observed-behavior does not match the stated-behavior.

If the program's processor-spec-graph is animated by an interpreter-active-processor whose observed-behavior matches the stated-behavior of the base-case-interface at the base of the processor-spec-graph then the bug corresponds to the derived-behavior of the program-code processor-specification-graph not matching the stated-behavior of the program-specification base-case-interface.

A programmer must then perform derivations of the behavior of the code, on the specific inputs which cause observed-behavior to differ from stated-behavior, until they find which step in a derivation causes the derived-behavior to differ from the stated-behavior. The bug is the portion of the internal-pattern of the processor-spec which caused the derived-behavior to differ from the stated-behavior.

2.1.4 The processor-specification interface

The interface portion of a processor-specification states the interface-elements that a higher-level processor-specification can use to specify its internals. The interface also states constraints on the relation of the interface-elements to each other.

For example, a programming language states its syntax, which specifies the set of acceptable interface elements. Meanwhile, the grammar rules state constraints on the relationships among the elements. Certain elements can only appear in certain relations to certain other elements. In C, the assignment oper-

ator '=' must be separated by a semicolon from another '=', this is a constraint on the relation of '=' elements to ';' elements and other '=' elements. Grammar rules state a large number of such constraints compactly.

The interface-elements are used to form *grammatic-sentences*. A grammatic sentence has one interface-element which is a command-name, and a number of *datum* syntactic-elements to which the command is applied. A datum is defined by the interface. It may be a single syntactic-element of the interface, such as an integer, or it may be another full or partial grammatic sentence.

The remainder of this paper states syntactic-entities in "text" form only. Each piece of syntax has a physical form. For example, a silicon-chip is an interpreter-internal form of processor-specification that uses physical structures – transistor-structures and wire-structures – as each syntactic-entity of the processor-spec. Meanwhile, an application program in source-form on disk associates a physical region of disk surface with each syntax-symbol in the program. The pattern of magnetic alignments within the region is the physical form of the syntax-symbol. The remainder of this paper uses whatever physical form of syntactic-entity that the reader is observing this paper in (toner on paper vs light-and-dark-dots on a screen). The physical form of syntactic-entities is not further taken into consideration.

The remainder of this paper also only considers commands whose relations go a single direction, with inputs distinct from outputs. IE, humans can apply the "multiply" relation backwards to deduce what the other input must have been in order to get a particular product, when given one of the inputs. This kind of "backwards" derivation from "forward" command stated-behavior is not considered in this paper.

2.1.5 Scheduling-constraints

The term *scheduling-constraint* must be defined in preparation for defining the internals.

This framework has two types of constraint: interface-constraints and scheduling-constraints. Interface-constraints determine what is a grammatic-sentence and what is not. Scheduling-constraints determine which grammatic-sentences may be scheduled and which may not.

Precisely stated, a grammatic-sentence is a grouping of syntactic-entities within a received-pattern. A grammatic-sentence is a command plus a number of datums that the command is applied to. The interface-definition states the number of datums that are grouped with each command-name. The interface-definition also states the syntactic form of each element of a grammatic-sentence, and the relative placement in the syntax-string-representation of the received-pattern of each element of a grammatic-sentence. Constraints that are stated in the received-pattern do not appear in any grammatic-sentences; they are handled separately.

Further, a *schedulable-grammatic-sentence* is a grammatic-sentence which satisfies all scheduling-constraints.

The kinds of constraints:

- *interface-constraints* – determine what is a *grammatic-sentence* and what is not, and determine which element is the command-name and which elements are datums. They appear only in interface-definitions, and are constraints, but not scheduling-constraints.
- *availability-constraints* – determine which datums are available for a given command-name to be applied to. More precisely, they determine which datums are candidates for inclusion in a schedu-

lable-grammatic-sentence containing a particular command-name. They are one type of scheduling-constraint and may appear in received-patterns or may appear in interface-definitions.

- *grouping-constraints* – determine the valid ways to pick from the candidate datums to form a single group that a command is applied to. More precisely, the availability-constraints first determine candidate schedulable-grammatic-sentences plus a pool of candidate datums for each. Then, for each candidate schedulable-grammatic-sentence, grouping-constraints determine which datums from the pool of candidates may appear together in that schedulable-grammatic-sentence. They are the other type of scheduling-constraint and also may appear in received-patterns or in interface-definitions.

As an example of a grouping-constraint, in Dataflow, when data arrives at one port of a command, it is constrained to only be grouped with the data arriving in the same "time-slot" at the other ports of the command. More precisely, the 50th datum at port A may only be grouped with the 50th datum at port B.

For another example of constraints, in C, a procedure-call appears as the procedure-name followed by '(' then a comma-separated list, then ')'; this syntactic-form is an interface-constraint – it says which syntactic-entities are part of a grammatic-sentence – the procedure-name as the command-name, and each position in the comma-separated list as each datum. The C language base-case-interface further constrains the elements appearing in the comma-separated list to be "evaluated" before the procedure call, which means each position must be in a language-defined form before the procedure-call takes place; this is an availability-constraint – only datums in a certain syntactic-form are available to a schedulable-grammatic-sentence that contains a procedure-call as the command-name.

2.2 The internals of processor-specifications

A processor-specification has ten internal elements: receiver, matcher, scheduler, extractor, lookuper, handoffer, insertor, returner, working-pattern-holder, and lookup table.

Here is a list of the internals and the stated-behavior of each:

- Receiver – gets a received-pattern from the handoffer of a higher-level active-processor, and puts it into the working-pattern-holder. A received-pattern is valid if it conforms to the active-processor's interface.
- Working-pattern-holder – holds a working-pattern while the pattern is being reduced. The receiver places received-patterns into the working-pattern-holder, the extractor takes scheduled grammatic-sentences out, the insertor puts the results of the grammatic-sentences back in, and the returner takes everything out of the working-pattern-holder.
- Matcher – identifies grammatic-sentences within a working pattern
- Scheduler – determines which candidate grammatic-sentences satisfy all scheduling-constraints, and so qualify as schedulable-grammatic-sentences. It then picks schedulable-grammatic-sentences and picks a lower-level active-processor to perform applying the command to the data. A scheduler is stated to make these choices non-deterministically. Grammatic-sentences that have been chosen by

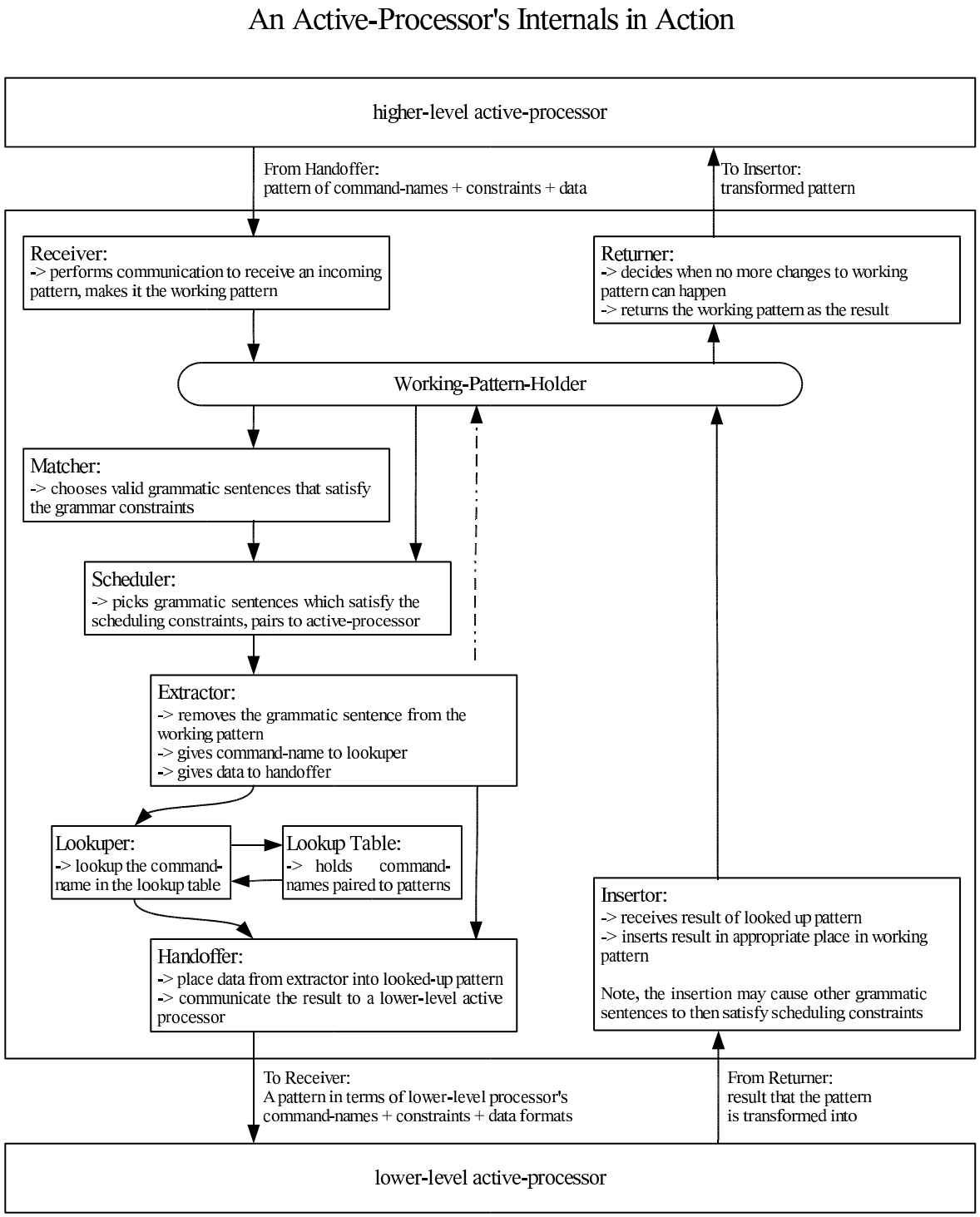

Figure 5. The flow of information among the internals and what actions each internal takes in response. Each internal is represented by a box. The solid arrows represent the flow of information. The dashed arrow represents one internal causing a change inside the working-pattern-holder.

the scheduler are called *scheduled-grammatic-sentences*.

- Extractor – for each scheduled-grammatic-sentence, removes the elements of the sentence from the working pattern, gives the command-name to the lookuper and the datums to the handoffer.

- Lookuper – finds a command-name in the lookup-table and retrieves the pattern associated with it. It gives the pattern to the handoffer.
- Handoffer – places the datums received from the extractor into the pattern received from the lookuper. It then hands the result to whichever lower-level active-processor the scheduler chose.
- Insertor – takes the result of the handed-off pattern from the lower-level active-processor. It inserts the elements of the result into the appropriate places in the working-pattern. The appropriate place is defined by the stated-behavior of the interface. An insertion may cause previously unsatisfied scheduling-constraints to become satisfied, thereby creating new schedulable-grammatic-sentences for the scheduler to choose from.
- Returner – when it detects that no scheduled-grammatic-sentences are outstanding, and that no schedulable-grammatic-sentences exist in the working-pattern, the returner removes the working-pattern and returns it as the result. This frees up the working-pattern-holder for the receiver to place new received-working-patterns into.

This choice of internals is arbitrary. This framework chose these particular internals with this particular assignment of function to each, but other choices are equally valid as long as they can be used as a base-case-interface in whose terms a processor-spec-graph can be stated that has derived behavior that matches the observed-behavior of computer-science's practices.

A side note, previewing a subject covered in depth in subsection ⟨reference|⟩, is that each internal element may be specified in terms of a Turing Machine, however, any sub-set of internal elements cannot produce observable behavior that requires a Universal Turing Machine. All the internal elements must be present in an active-processor in order for it to display the kind of observed-behavior that can only be computed with a Universal Turing Machine. More precisely, given a processor-specification-graph, one attempts to form a second processor-specification-graph which has some form of Turning Machine as its base-case-interface such that the derived-behavior of both specification-graphs is the same. If this is only possible when using a Universal form of Turning Machine as the base-case-interface of the second processor-specification-graph, then all internal elements must be present in the active-processor which is an animation of the first processor-spec-graph.

Processor-specifications often rely upon the interpreter-active-processor to supply most of the internals. In many languages, processor-specs built upon the language base-case-interface cannot specify any internals other than the lookup table. Hence, the internals missing from the processor-spec are supplied by an interpreter-active-processor exhibiting observed-behavior that matches the stated-behavior of the language base-case-interface.

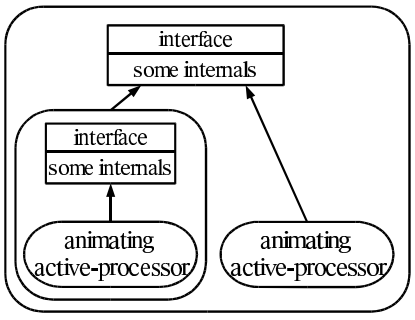

Figure 6. The animating active-processors at the bottom are both interpreter-active-processors which supply the internals that are not stated in the processor-specs.

2.2.1 Optional Internals

One optional internal not shown in figure 5 is the Creator. This internal is used to create a new activation. Part of the process of animating a processor-spec-node is creating a new activation which has the interpreter-internal-form of the processor-spec-node and has the initial-state of the processor-spec-node.

Another optional internal not shown is the switcher. This internal is what allows a single base-case-active-processor to animate several processor-spec-nodes. In this way, a single base-case-active-processor becomes multiple active-processors. This internal stops the matcher and scheduler, communicates an activation (including current state of working pattern) to a memory-processor, gets a different activation from a memory-processor, then re-starts the matcher and scheduler on the new activation (including new working pattern).

For example, a Java thread-object is an activation. The JVM is the base-case-active-processor. The thread-object exposes the activation internal to it. The thread-scheduler is the Switcher, and the `new` command on a runnable class is a command to the JVM's internal Creator.

Some readers will have noticed that all categories of command are under the control of the scheduler, including the Creator and the Switcher, lookup-table-modification command. The scheduler communicates with the Matcher, Extractor, Working-pattern, Creator, and Switcher.

2.2.2 The link between internals and processor-spec-graphs

Each activation has an interpreter-internal-representation of a processor-spec-node. The activation is the state of an active-processor-node. The receiver in the active-processor-node receives a pattern and places it into the working-pattern-holder. For example, say the processor-spec-graph is a C program.

The incoming pattern consists of procedure-names, language-command-names, variables, constants, and scheduling constraints. The matcher picks out grammatic-sentences, for example, it finds a procedure-name plus the expressions in the parameter-positions. One of the expressions is an array-access command ("`var1 = myProcedure( array[var2], var3, 1);`"). Of these, the `var2` and `var3` are the only grammatic sentences in schedulable form. The language specifies left-to-right precedence, which is an implied constraint, so the only schedulable grammatic-sentence is `var2`. This is chosen by the scheduler, which is INSIDE the base-case-active-processor, so it knows that the command (implied lookup) is an element of the base-case-interface. The base-case-active-processor's extractor pulls the `var2` out of the working-pattern, gives it to the lookuper which gets a pattern for the memory-processor, gives the pattern to the handoffer, which puts the `var2` with it and hands it off to the memory-processor. The memory-processor gets the `lookup var2` pattern, looks it up in its lookup-table and returns the result. The insertor in the base-case-active-processor gets the memory-processor's lookup result and inserts it into the working-pattern where the `var2` used to be.

As far as the C program is concerned, the `var2` memory-lookup is performed as a primitive of the base-case-active-processor. We choose to model the C-base-case-active-processor as handing off to a separate memory-processor.

Notice that the same scheduler, embedded in the base-case-active-processor is scheduling commands from several different activation's lookup-tables. Likewise, the rest of the internals are re-used on several different active-processors.

Interestingly, notice what happens when the processor-spec defines its own internals. Consider a C program which parses a command-line command then takes the specified action. This program defines its own working-pattern-holder, matcher, scheduler, and returner. Notice that not all internals are defined in the C code. In particular, it is an OS mechanism that places the command-line into a buffer and puts the buffer in an activation of the program. The base-case-active-processor is created by the the OS-active-processor and begins existence with the command-line-pattern already in the working-pattern-holder. When the program is done, it returns a result, usually an "exit-code". The returner is once again part of the OS-active-processor, which the C-active-processor uses.

The C program implements a parser, which is the matcher. It uses a `switch` statement to jump to code that performs the actions of the command. The `switch` statement acts as scheduler, extractor, lookuper, and handoffer. The target of the switch is then the next level down processor-spec-node. Keep in mind that multiple levels of processor-spec-graph may be stated in the same processor-spec-unit, which is a single file in the case of C. The nodes of the processor-spec-graph are determined by scoping rules and lookups. Any code which is the result of a lookup is received by a lower-level processor-spec-node than the node which performed the lookup, by definition of processor-spec-node.

A subtlety at work is that the base-case-active-processor has the C program as an input-pattern, and performs the action of the `switch` statement. Thus, one active-processor's internals produce observed-behavior that is the steps of a higher-level active-processor's internals. Every active-processor has all internals. The point of confusion is that not every processor-spec must state all internals. Every processor-spec states at least one internal. The missing internals are filled in by the base-case-active-processor. The internals that are stated are given observed behavior by the base-case-active-processor giving behavior to the commands the processor-spec's internals are stated in terms of.

The result is that when viewing code through the lens of this framework, one must keep in mind which internals are defined in the code and which are inherited from the language.

2.2.3 Stating a Base-Case-Interface

A base-case-interface has the following things:

1. A set of syntactic entities
2. An active-processor that holds the source-form processor-specs that are stated in terms of the base-case-interface. The specification of this active-processor provides the "scaffolding" for source-form processor-specs. It gives meaning to the words "position" and "relation" as in "one syntactic-entities position relative to another syntactic-entity". As another example, grammar constrains position on line where a command-name can occur, and where a datum can occur. Here, "line" is an interface-element of the active-processor that holds the syntax, and value of "position" has to be asked for from that active-processor that holds the syntax. The very physical embodiment of the syntax is defined by the active-processor holding the syntax.
3. A grammar which states allowed relationships among syntactic-entities. "Relationship" is defined by the active-processor that holds the source-form of processor-specs stated in terms of the base-case-interface. This is stated in terms of some other base-case-interface (such as BNF).

4. Syntactic form of command-names. This is stated in terms of some other base-case-interface (such as BNF).
5. Syntactic form of data. This is stated in terms of some other base-case-interface (such as BNF).
6. Stated behavior of each command-name, on the data taken by that command-name. This includes what syntactic-entities are removed from the working-pattern, and what syntactic-entity(ies) replace them. This is stated in terms of some other base-case interface (such as some formal semantics, or natural language).

A base-case-interface's stated-behavior is always stated in terms of one or more different base-case-interfaces. For example, the document stating the commands of an application, the format of the commands and data, the grammar of them, and the behavior of each command is stated in terms of natural language, or an object-model language, or drawn pictures, and so on. Natural language is a base-case interface, as is an object-model language, as are pictures in which each element matches some pattern which has some behavior stated elsewhere, and so on. In fact, those base-case-interfaces are then themselves defined in terms of other base-case interfaces. No "bottom" base-case-interface exists.

3 Viewing Things in Computer Science Through This Framework

Now that the basic elements of the framework have been defined and illustrated, we turn our attention to using the framework as a base-case-interface. We wish to build processor-spec-graphs which correspond to portions of computer science. With these we want to show that the derived-behavior of the processor-spec-graphs corresponds well to observed behavior of the practice of computer science.

In other words, we want to show that this framework fits the practice of computer science. We do this for three reasons: 1) we want to increase the reader's confidence in this framework; 2) we hope this gives "aha"s about computer science; and 3) we will build upon this to show in the next section what a language should include in order to enable hardware independence across a wide range of hardware.

3.1 Scheduling Constraints and Programming Languages

Scheduling constraints can be categorized by "where" their definition is stated:

1. *program-defined-constraints* – the behavior of a constraint is stated in the same processor-spec-unit set as the grammatic-sentences that are constrained.
2. *base-case-defined-constraints* – the behavior of a constraint is stated in the same statement of the behavior of the base-case-interface elements. A program would then be a processor-spec which is stated in terms of the base-case-interface's elements.
3. *externally-defined-constraints* – their behavior is defined somewhere external to the program. For example, while writing a thread-based program, a coder may draw pictures which represent the constraints on communication between physical-processor-nodes. These pictures define the scheduling-constraints in terms of a base-case-interface that the coder has made up for themselves. They then state a pattern of locks in the code such that the observed-behavior of the animated code never violates the scheduling constraints that were in their pictures (or maybe only ever in their heads).

3.2 Parallelising Compilers

Parallelising compilers make scheduling decisions. Some scheduling decisions choose which data is included in a particular grammatic-sentence and assigned to a particular active-processor. These decisions may, for example, break up a matrix into sub-matrices and assign each sub-matrix to a different physical processing-node.

In order to make these decisions, the compiler must be able to calculate whether a proposed decision will allow the same derived-behavior to result from the compiled program as from the source-form of the program.

Only three things determine derived behavior:

1. The constraints on forming grammatic-sentences
2. The behavior of grammatic-sentences
3. The constraints on scheduling grammatic-sentences, both choosing which to schedule, and choosing which active-processor to assign a chosen sentence to.

The parallelising compiler has been written with the stated-behavior of the base-case-interface of the program available. Thus, the compiler has the information needed to be sure that once a grammatic-sentence has been scheduled that the observed-behavior of that sentence corresponds to the stated-behavior. The compiler just needs the scheduling constraints in a syntactic-form that the compiler can use.

However, only languages which allow program-defined-constraints allow a program to provide a compiler with a syntactic-representation of the constraints. Languages which provide base-case-defined-constraints require the compiler-programmer to place syntactic-representations of the constraints into the processor-spec-graph that states the source-form of the compiler. Languages which provide only externally-defined-constraints have no means to communicate scheduling constraints from the programmer to the compiler. Instead, the parallelising compiler must infer scheduling constraints.

Inferring scheduling constraints is a search process which is most likely NP-hard. Current approaches include pattern-matching to code-patterns for which a syntactic-form of scheduling constraint has been placed into the compiler-processor-spec-graph. In general, to find all scheduling constraints, the parallelising compiler must try various modifications to the program and various grammatic-sentence-to-physical-processor choices, run each on a data-set that is guaranteed to expose all observed behavior that differs from stated program-behavior, collect all modifications which preserve behavior, and infer scheduling constraints from that set of behavior-preserving-modifications and scheduling-choices. In practice, this process is likely to take longer than the useful life of the program.

3.3 Lambda Calculus stated in terms of this framework

For convenience, we choose Benjamin Pierce's concise definition and notation of Lambda Calculus in his book covering type-systems ⟨cite | Pierce2002⟩ ("Types and Programming Languages", Benjamin C Pierce, MIT press, 2002, pp 51-73). The operational semantics of Lambda Calculus for call-by-value are:

$$t ::= x \mid \lambda x.t \mid t\ t \qquad v ::= \lambda x.t$$

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad \frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'} \qquad (\lambda x.t_{12})\ v_2 \to [x \mapsto v_2]t_{12}$$

The semantic rules give the stated-base-case-behavior, as well as giving scheduling constraints.

The pure Lambda Calculus has no command-names. Therefore it has no lookup-table. However, it still has a lookuper, handoffer, and insertor. Here's how it works:

1. The entire program is the received-pattern placed into the working-pattern-holder during creation of the activation.
2. The matcher works as normal. The syntax for terms and values tell the matcher how to identify grammatic-sentences within the working-pattern. A grammatic-sentence is of the form "*t t*". Only grammatic-sentences which are of the form on the left-side of the arrow below the line are schedulable-grammatic-sentences.
3. The scheduler only gets at most a single schedulable-grammatic-sentence at a time (due to the combination of rules which enforce call-by-value scheduling). When one appears, the scheduler passes it along to the extractor.
4. The extractor removes the grammatic-sentence from the working-pattern. It hands the left-hand term to the lookuper and the right-hand term to the handoffer.
5. The lookupper simply drops the lambda, extracting the body. It then tells the handoffer the locations in the body-pattern that it should insert the parameter.
6. The handoffer inserts the right-hand term it got from the extractor into the locations pointed to by the lookupper, then gives the result to the insertor. It performs no handoff to any other active-processor.
7. The insertor puts what it got from the handoffer back into the working-pattern in the place the extractor tells it to.
8. The matcher now has new stuff to try to find a schedulable-grammatic-sentence within. If none is found, then the returner detects this and returns whatever is in the working-pattern-holder as the result.

In practice, richer forms of lambda-calculus are used which do have lookup-tables. The handoffer in these has dual behavior, being able to operate as described here and also operate as shown in section 2.2.

3.4 Defining Categories of Parallel Programming Languages

The three "places" that scheduling constraints get defined suggests a means of categorizing programming languages:

1. Languages that provide scheduler-implementation commands that are used to implement schedulers which don't violate externally-defined-constraints
2. Languages that provide base-case-defined-constraints
3. Languages that allow program-defined-constrains

Languages in each category:

1. externally-defined-constraints: Java, C with threads, MPI, Linda, (? portions of X10, Titanium, and ZPL? – in ZPL, have to state how divide data.. same for X10.. and HPF has constructs for preference of how to divide data..)

2. base-case-defined-constraints: HPF, ZPL, base C without a thread-library, Sisal, X10, Titanium (check X10 carefully)

3. program-defined-constraints: research languages which provide guards, languages with the "Atomic" declaration but no locks; the CodeTime platform's BaCTiL

3.4.1 Sequential languages

First, we discuss the "base" sequential languages that have no parallelism exposed by the language-defined base-case-interface. In these languages, base-case-defined-constraints are used by the coder to constrain the order in which grammatic-sentences are scheduled. Examples of program-stated but base-case-defined-constraints are ';' , '()'. Examples of base-case-defined constraints which are *not* stated in the program, but are instead implied are order of precedence, and the "call-by" semantics (for example, call-by-value semantics). A serial language includes enough of these in the base-case-interface it defines that every program-processor-spec-graph has one and only one set of valid scheduling decisions which can be made over the course of the run of the program.

Some readers may be wondering about scheduling program-defined commands. In sequential languages, conditional statements are used in the construction of schedulers for program-defined-commands. For example, switch statements are often used to pick which task is performed, say based upon some input-pattern such as a character from the keyboard. This is an example of externally-defined-constraints. The sequential statements are used to implement a scheduler whose behavior is consistent with external constraints on the scheduling of program-defined commands.

3.4.2 Externally-defined-constraint languages

Parallel languages in this category have commands specifically for *implementing* schedulers that can schedule overlapping work-units via causing handoff to distinct active-processors. When a scheduler is implemented by these types of commands, no syntactic form of the constraints appears in either the program or the base-case-interface. Only the programmer has a syntactic form, either in their head or perhaps written down on paper in some notation they have made for themselves (the notation is a base-case-interface for scheduling constraints that they have defined for themselves).

Java's `lock` and `synchronized` statements are used to implement constraints on which program-defined commands can overlap execution with which other program-defined-commands. A critical section is a program-defined command. The synchronized keywork causes observed behavior that end up conforming to an externally-stated constraint. Likewise semaphores, monitors, memory barriers and similar statements in other languages and libraries are used to implement schedulers whose observed-behavior matches externally-defined constraints (if the coder is lucky).

Titanium scientific Java things that implement scheduler

Communication primitives in Communicating Processes –(pg 303 to 316 of "The Formal Semantics of Programming Languages" by Glynn Winskel, 2001) this is π-calculus, Hoare's CSP, Milner's CCS, and related formalisms.

3.4.3 Base-case-defined-constraint languages

DoAll loops, as in HPF, and other "parallel constructs" added to sequential languages

X10 things that implement scheduling constraints

Titanium

ZPL things like the array-index-var and whatnot

3.4.4 Program-defined-constraint languages

We are unaware of any widely used languages which have statements to declare program-defined-constraints. However, theoretical models described by Dijkstra and by Hoare <**double check this, get citation right**> define guarded commands. The guard is a boolean; if it is true then the scheduler can schedule the associated command; if it is false, the scheduler is not allowed to schedule the associated command. The syntax which indicates the presence of the guard plus the syntax of the boolean states the scheduling constraint. Thus, guards are program-defined-scheduling-constraints.

A more recent construct appearing in the literature is the `atomic` declaration. The semantics vary, however, the appearance of the `atomic` statement does mark a program-defined-constraint if that appearance controls the scheduler. For example, if marking a section of code `atomic` is what causes the scheduler to not overlap executions of that code-segment, as opposed to being an annotation for use by a theorem prover, then the `atomic` statement plus the code it marks state a program-defined-constraint.

The CodeTime platform's BaCTiL language ⟨cite | BaCTiLTechRep⟩ has the most general form of program defined constraints that we know of. A program is divided into atomic units. Each unit is guarded by a boolean which selects data from multiple memory-processors. The guards decide which memory processors may be grouped together in a schedulable-work-unit for the guarded atomic-code-unit. Any data in a memory-processor may be used in the guard's boolean, and each group which satisfies the boolean may be scheduled at any time, including overlapping. Multiple scheduled-work-units of the same atomic-code-unit may overlap because atomic-code-units only have temporary variables. Bookeeping such as loop indexes is carried with the data. The means that the language has no notion of time. Instead, if the algorithm requires causality, the programmer generates some appropriate form of data to represent time and then uses it in a guard.

Some readers may have noticed that this implies that BaCTiL programs have many separate memory processors created and destroyed during a program-run. An even greater number of active-processors are created and destroyed, one for each scheduled-work-unit. This contrasts to both guards, as defined in Winskel ⟨cite | ⟩, and `atomic`, whose semantics imply only a single memory-processor, and a single active-processor whose lifetimes span the entire program-run.

Atomic says "no overlap of scheduled-work-units that contain the same atomic-marked code". What it often implies, however, is that no other work-units that communicate to the same memory-location-names as those in the marked code may be scheduled overlapping. In other words, there is really an externally-defined constraint on locations in the memory-processor (a location in a memory-processor is a name in the memory-processor's lookup table) which the `atomic` statement is used to implement.

3.4.5 A detailed look at Guards

Guards are a form of program-defined-constraint. The presence of a guard indicates that a scheduling-constraint exists which affects the conditions under which the guarded pattern can be scheduled. The boolean-pattern defines the constraint itself.

We will show in sections ⟨reference|⟩ and ⟨reference|⟩, that the inclusion of BaCTiL's type of program-defined-constraint enables hardware-independence by giving needed information to compilers and *run-time-systems* (a run-time-system is a collection of active internal elements, sometimes all of them, but at the least an active-scheduler).

A guard, as defined in Winskel (pg 298 to 303 of "The Formal Semantics of Programming Languages" by Glynn Winskel, 2001), is a boolean that is paired to a command. The pairs appear in lists. Given a state, any pair whose boolean evaluates to true on that state may be scheduled. The scheduling is non-deterministic. (In Winskel, only one scheduling-event per state is allowed; only one pair whose boolean is true on a state may be the one and only pair scheduled on that state, however this is an unnecessary restriction). The result of a scheduling-event is a new state, on which the process repeats. (In Winskel, semantics imply state -> schedule -> result -> new_state is an atomic unit.. only one scheduled-work-unit outstanding at a time).

One way to model this in our framework is with multiple processors. One or more active-memory-processors hold the σ. One active-processor presents the base-case-interface. However, the active-processor which presents the base-case-interface may be the result of multiple active-processors collectively animating an interpreter-processor-spec-graph.

A simple way is the interpreter processor-spec-graph to contain a separate scheduler-internal-element processor-spec-graph which is animated by a single active-processor, such as a silicon-active-processor. This active-processor performs the behaviors of many of the internal-elements and hands-off scheduled-work-units to a pool of other silicon-active-processors.

A more complex model is a peer-to-peer model in which multiple active-processors present the base-case-interface together. They collectively animate an interpreter-processor-spec-graph which is then the active-processor that animates a single guard-based processor-spec-graph. They must collectively animate a scheduler-processor-spec-graph as a single scheduler-internal-element active-processor. This implies that the active-processor-graph contains loops, by which they handoff to each other as well as receive from each other. The working-pattern and several of the other internals must also be collectively animated.

Each peer asks a memory-active-processor to perform lookup in σ, the result of which is inserted into the collective working pattern and may cause one or more booleans to become true. The scheduler-animating active-processors collectively decide which pairs to schedule and which of the scheduler-animating active-processors will switch over to animating a command-reduction processor-spec-graph and which looked-up pattern of a guarded command each of those command-reduction animating active-processors gets.

The take-away from this discussion of guards is that they are a form of declarative scheduling constraint, which means the constraint is defined in the program along with the program-defined-commands that are constrained.

4 Creating Hardware-Independent Programming Languages

Here we outline our ideas about the underlying requirements of a hardware-independent programming language, then we use the framework to state some ideas of features a language should have in order to satisfy those requirements and enable hardware-independence. However, achieving hardware-independence requires more than just language features. A more detailed discussion of the requirements for hardware-independence, high productivity, and wide acceptance is given by Halle ⟨cite|Dependence⟩.

4.1 What is required to tune a program to specific hardware

Hardware-independence means high performance of a single *distribution* on each different hardware platform (a distribution is the form of a program that gets installed on a machine, for example what is on the CD-ROM used to install a software application). This requires some means to tune the distribution when it arrives on a particular hardware platform. The scheduler is responsible for assigning work-units to physical-processing-nodes, so it is the entity that is responsible for the performance of a specific application on specific hardware. Therefore, we assign the scheduler the task of tuning a distribution. We expect part of the scheduler of the base-case-active-processor to be in the compiler, and part to be in a run-time.

4.1.1 A Scheduler's goal

A scheduler's goal is to either maximize throughput of hardware, or minimize start-to-finish time of programs, as measured by one or more quantities:

- Total wall-clock time from start to finish of a given program-run
- Throughput of a machine during a given wall-clock period, or average wall-clock period
- Wall-clock delay from user request to results available
- Percent of resources kept busy, during a single run, or averaged over a wall-clock period
- Percent of resources consumed by the scheduler
- Percent of start-to-finish time added due to scheduler think-time in the critical path

4.1.2 What knobs a schedular can adjust to achieve its goal

A scheduler can adjust certain quantities in order to achive its goal:

- comp-to-data ratio of code-units
- size-of-data that is paired with a code-unit to form a scheduled work-unit
- frequency of making scheduling decisions
- complexity of each scheduling decision
- order of handing-off scheduled work-units
- timing of handing-off scheduled work-units
- the resources consumed by each scheduled work-unit (processor-node, network links, and so on)

4.1.3 What info a scheduler needs to enable it to adjust the knobs

In order to decide which knobs to adjust, and by how much, the scheduler needs to know:

- comp-to-data ratio of code-units
- constraints on combining code-units into larger code-units, which will have higher comp-to-data
- constraints on dividing data
- steps to take to divide data
- dependencies among schedulable-work-units
- slack of each schedulable-work-unit (slack == max delay in start time without incr. critical path)
- which unfinished work-units are at what physical locations (what resources are spoken for)
- wait time at each physical-processing-node (what work is in buffer)
- what data is at what physical locations (physical location is one separated by comm channel)
- BW, utilization, latency, and burstiness of communication channels
- comp-rate of physical-processing-nodes
- predicted consumption of resources by a proposed scheduling of a work-unit (how long will run)

Mainly, a scheduler has to not waste too many resources in the making of decisions, while tuning the size and placement of work-units to make maximal use of the hardware resources, where maximal is either throughput or latency.

One consideration on scheduling overhead is responsiveness to changing load and resource availability. A heuristic is to make scheduling decisions spaced in time on the order of the system response-time (on the order of the network latency). Faster does not improve scheduling decisions much because little information changes between decisions. Slower does not respond to changes in the system fast enough to keep all resources busy. Thus, the run-time of each scheduled work-unit would be tuned to be on the order of the system-response time, and a scheduling decision is made each time a work-unit completes.

The most crucial knobs the scheduler can turn are the comp-to-data ratio and the size-of-data of a schedulable work-unit. These are the main quantities which affect how well a program runs on particular hardware. In order to turn these knobs, the scheduler must know the constraints on combining code-units and constraints on dividing data. The fewer of these constraints that are stated in the program and stated in the base-case-interface, the more choices the scheduler has to tune work-unit run-time to match system response-time, which we argue gives the point of highest performance.

4.1.4 Minimum constraints

We use the notion of proto-algorithm ⟨cite | ⟩ to discuss minimum constraints. By definition, the proto-algorithm states the minimum constraints underlying any implemented algorithm. In other words, the same proto-algorithm can be implemented in C and in HPF. The C implementation will state many more constraints (constraining to pure sequential) than the HPF implementation, even though both implement the same intuitive-notion-of-algorithm. Proto-algorithm gives a precise definition of the intuitive-notion-of-algorithm.

The discussion of language categories shows that some languages allow an implementation to be closer to the proto-algorithm. In particular, implementations in sequential languages typically have the most extraneous constraints and so are the furthest from the proto-algorithm, followed by implementations in base-case-defined-constraint languages, then externally-defined-constraint parallel languages, with program-defined-constraint languages allowing the closest match between implementation and proto-algorithm. In other words extraneous unneeded constraints get included in implementations when the language doesn't allow fully-general declaration of scheduling-constraint form and placement.

Externally-defined-constraint languages allow a coder to create a scheduler whose behavior obeys the minimum constraints stated by a proto-algorithm. However, such a scheduler is so difficult to create and get right that in practice extraneous constraints are introduced in order to make the programming task easier. More importantly, the scheduler implementation encodes choices of size-of-data, and choices of code-unit specific to particular hardware. The scheduler implementation is part of the application implementation. Thus, the application source code must be modified when hardware changes significantly. Because the constraints are externally-defined, the base-case-active-processor's scheduler does not have a syntactic form of the constraints, so it cannot make the adjustments automatically. We saw this same problem from lack of syntactic form of constraints in the discussion of parallelising compilers.

On the other hand, a language which has general program-defined-constraints allows programs with the fewest total constraints. More importantly, it communicates a syntactic form of those constraints directly to the base-case-active-processor's scheduler. The base-case-active-processor's scheduler can then turn the knobs for comp-to-data and size-of-data without modifying the application source code.

Based on this, we argue that a hardware-independent language should include general program-defined-constraints. This allows expressing the fewest possible constraints, which enables the widest range of comp-to-data ratios and sizes-of-data. It also gives a syntactic form of the constraints directly to the scheduler, which the scheduler needs in order to adjust the comp-to-data and size-of-data to specific hardware without human intervention.

4.2 What features a hardware-independent language would benefit from

As we just argued in section 4.1.4 that a hardware-independent language should include general program-defined-constraints. This enables the base-case-active-processor to decide how far to turn the knobs, which goes a long way toward separating hardware from applications. However, there still must be a mechanism to divide up application data and re-combine results of the pieces. Also, every program exists in an environment, so an interface should be available to isolate the application from the hardware details of the environment.

4.2.1 Separating Application-knowledge from Hardware-knowledge

In order for an application-program to be independent of hardware, it must have no embedded assumptions, constraints, or other information related to the hardware the application may run on, by definition of hardware-independent. Thus, the language in which the application is written should have no features which make assumptions about or expose knowledge of hardware.

By their nature, fixed parallel structures added to languages come with assumptions about hardware those structures map well onto. Even worse, implementing a custom scheduler as part of an application embeds knowledge of the hardware into the application.

However, if the only constraints stated or implied in an application are those of the proto-algorithm, then the minimum assumptions about hardware have been embedded into the application. Thus, the language won't introduce any extraneous hindrances to hardware-independence.

4.2.2 Install-time compiler

Program-defined-constraints communicate constraints in syntactic form directly from the coder to the scheduler inside the base-case-active-processor (which we expect is split between a compiler and a run-time system). This lets the scheduler decide the size of code-units and size-of-data for particular hardware, however, a mechanism is still needed to perform the changes.

We propose a back-end compiler which runs at install-time to perform the code-unit size changes. Install time is the earliest point in the application life-cycle at which hardware characteristics are known. It is also acceptable for the installation process to take a fairly long time because it is performed once and amortized across all program runs on that hardware. This should be palatable even to home and business users for their personal machines if the install-process collects configuration information then performs the rest of the install autonomously.

This compiler has a time budget which allows sophisticated optimizations such as changing the size of code-units, tuning machine-code to the particular memory-heirarchy of physical-processor-nodes, and fitting the particular pipeline structure of the physical-processor.

An install-time compiler isolates distributions from hardware, while still allowing each hardware-platform to have its own compiler and own run-time scheduler. Interestingly, this makes a software distribution agnostic to physical-processor instruction set. This frees processor architects to change instruction set in each generation, and thereby explore many techniques currently un-practical.

4.2.3 Divide-body-undivide pattern

For size-of-data changes, we propose the divide-body-undivide pattern. It interfaces an application-provided "black box" to the base-case-active-processor's scheduler. The box performs division of application data into sizes the scheduler tells it are optimal for the hardware. The BaCTiL language incorporates this pattern ⟨cite|⟩.

The idea is to expose the scheduler as an explicit entity in the programs. Data is given to this scheduler-entity, packaged in a generic envelope that the scheduler expects. The scheduler then decides what size of pieces (or how many pieces) it wants to break that data into. It adds this decision to the envelope then hands the enveloped-data to an application-provided divider, which performs the division and hands the pieces, each in its own envelope, back to the scheduler. The scheduler may then choose to repeat the process on any or all of the pieces. When it is satisfied, it passes the pieces on to an application-provided body, then from that to an application-provided un-divider which re-combines the individual results into the overall result.

In this way, a clean interface is provided between application and hardware. The divider, body, and undivider are application specific. The scheduler is hardware-specific. The scheduler-entity in the program is the communication mechanism between the two.

4.2.4 Hardware-independent OS

A language should have an interface to an abstract operating-system that remains the same across all hardware. The BaCTiL language, in conjunction with the CodeTime Virtual Server provide this; a paper on the language specification ⟨cite | BaCTiL⟩ and a paper on the Virtual Server specification ⟨cite | CTVS⟩ give more details. The OS interface should provide a name-space mechanism, a uniform interface to persistent data, a means to transform data between outside formats and internal formats, and a uniform means to discover the presence of desired persistent data and desired functions (such as printing, CD-ROM, visual display, and so on).

4.3 Programmer Productivity benefits

The proposed hardware-independence language-features also enhance programmer productivity.

Separating application concerns from hardware concerns reduces the complexity of writing applications. It also allows writing a program once and using the "dusty deck" for many years with no need for maintenance due to hardware changes.

Program-defined-constraints makes writing code easier. Mapping a proto-algorithm onto a fixed set of base-case-defined-constraints such as DoAll loops and matrices is challenging. Informal experience with undergraduate computer science students suggests that program-defined-constraints may be a more natural way for humans to express the constraints they have in their heads than matrices, DoAll loops, `points`, `regions`, `maps` and other fixed parallel-structures.

Implementing schedulers is notoriously hard. Once the number of locks rises above 6, even the most experienced programmers begin having difficulting getting multi-threaded code right. In general, implementing a scheduler requires predicting all the possible orderings of events in time, and controlling those with primitive mechanisms. The number of orderings increases exponentially with time for most programs, making this an exceptionally difficult task.

By having the base-case-active-processor's scheduler perform all the scheduling, the coder does not encounter the mapping-onto-fixed-structures problem nor the exponential-number-of-orderings problem. They also do not have to know anything about the hardware. Instead, each hardware platform has a small team of highly skilled programmers to make a scheduler once. This is then amortized over all programs ever written in the language.

Finally, knowing the constraints in a declarative form is a pre-requisite for both mapping onto fixed language structures and for implementing a custom scheduler, so simply writing down the declarative form avoids the extra work.

5 Conclusion

This paper has shown that a hardware-independent language should have program-defined-constraint features, should have a mechanism that lets the base-case-active-processor's scheduler divide application data, and an interface to a hardware-independent OS.

In order to show these things, this paper has introduced a framework that defines terms such as base-case-interface, active-processor, and scheduler.