The Elements of the CodeTime Software Platform

by Sean Halle

Website: codetime.sourceforge.net

Email: seanhalle@yahoo.com

Abstract

This paper describes the elements of the CodeTime parallel software platform. The platform's goals are to provide "write-once, compile once, run high performance anywhere" for parallel software, high programmer productivity, and wide-spread acceptance.

The platform includes three top-level components: a virtual server with a novel computation model, a family of source languages which compile down to that computation model, and a complete development environment.

The virtual server is the heart of the platform. It presents the appearance of a single entity, regardless of hardware underneath. The virtual server is a single name-space, and it is composed of a federation of systems. In this model, everything is a named system: each file, each OS function, each program, each server on another machine. A program takes the form of a mutable circuit. This enables both machine-independence and high performance.

The family of source languages includes a bare-bones language, and will later include both an ML-reminiscent language and an OO-flavor language.

The development environment includes: visual authoring, test harness, compiler, build tools, debugging, and team-development infrastructure.

1 Introduction

Writing software for High End Computing requires parallel programming. However, this type of programming is difficult. It is especially difficult to write both correct and high performance programs. Thus, programmer productivity is low, time-to-solution is long, and the number of proficient programmers is small. All of these factors combine to make parallel code expensive to write. Further, additional work must be done to make existing code run on a new machine. The complexity of the code makes this and other maintenance updates also expensive.

Significantly, all major microprocessor manufacturers are building either mult-threaded or multi-core chips for the next generation. The days of single-threaded processors are over. As the years pass, the number of threads on a chip will increase exponentially.

This requires commercial software developers to also write parallel programs.

However, no viable solution has yet been found for high performance hardware independent parallel software that has high programmer productivity and wide acceptance.

In another paper [*], the author argues that an integrated platform which includes all aspects of the software life cycle is needed to maximize the three goals: hardware independence with high performance, programmer productivity, and wide-spread acceptance in practice.

In this paper, the details of such a platform are presented. Other papers on the CodeTime platform include: one on the motivation for the choices of what to include[*]; a theoretical framework which helps to understand the new semantics introduced in the CodeTime computation model (alg defn[*], lang as processor[*], op semantics[*]); the experience of using the platform[*]; description of a high level language used to write code for the platform[*]; the operating system and security model[*]; and implementation details of a run-time system that implements the scheduling (control) portion of the computation model as a peer-to-peer system running on hardware made up of a number of processors connected by a network [*].

2 Background

Programs, languages, and physical processors can all be seen as the same thing: a set of commands, plus a scheduler which chooses input-sets to those commands. In this framework, a hierarchy exists with the user and "real world" at the top, cascading down through the program, the language, the processor, boolean-logic circuts, transistor models, down to basic physics. Each level is a processor whose commands are defined in terms of lower-level processor-commands. The schedulers for the higher-level processors are either custom or use the same scheduler as the lower level processor(s).

Languages historically fall into two categories: those that force the program to use the language's scheduler, and languages that provide primitives from which a program can construct its own scheduler. Parallel languages such as HPF, para-C, and Sisal fall into the former. They provide parallel constructs with relaxed scheduling constraints, forcing the program to use the language's scheduler. On the other hand, languages such as Java, pi-calculus, and Linda provide primitives such as locks, guards, and tuple-operations, from which a program may construct its own scheduler.

CodeTime's computation-model introduces a third alternative: modify the scheduler of the language itself. It provides both a "plug-in" mechanism and an extension mechanism.

The plug-in takes the form of programmer-supplied code that knows how to divide programmer-supplied data-structures. This divider is called via a standard interface from the scheduler during a program run. In this way, data may be broken into a hierarchy (tree) of arbitrary size pieces.

The extension mechanism allows the program to define custom scheduling constraints, which the language's scheduler then enforces. This allows the program to specify exactly the same constraints as the underlying algorithm, achieving the minimum set of constraints.

3 The CodeTime Approach

CodeTime works toward reaching the 3 goals by defining a platform. The platform covers all aspects of the software life-cycle, from authoring, to compiling from source, to distribution, to installation, to run, to maintenance.

The platform has three main components: a virtual server, a family of source languages, and a development environment.

The virtual server presents the abstraction of a single machine. No matter what hardware is underneath a given implementation of the virtual server, it appears to programs and people as a single entity.

That entity is a federation of systems. It has a single name-space, each system having its own name. Each program installed in the virtual server is a separate system. Each OS function is a separate system. Each file is a separate system.

A virtual server is defined as the collection of all systems in a single name-space such that all default systems are present (eg all OS functions), and all systems conform to the virtual-server specification.

A program is a circuit. When the program is installed, the virtual server makes that circuit a system. The virtual server gives that system a name. Other circuits can discover the name and use it to interact with the newly created system.

Program-circuits have function-units and system units. System units are interfaces to other systems (other programs, OS functions, hardware, etc). Function units perform the computation. They contain a single function, and "create" copies of themselves as needed. Thus the number of active circuit-elements is not fixed, but changes according to the evolution of the computation (the number of active circuit-elements extant at any given time is chosen by the circuit's scheduler). The virtual server implements this computation model.

One source language has been defined and two others are planned. The one defined is called BaCTiL (Base CodeTime Language). It is a simple language, analogous to C. BaCTiL is to the CodeTime computation model as C is to the Von Neumann computation model. In addition, an analog of ML is planned, called MeLCTiL, and an analog of object oriented languages is planned, called OCTiL ("Ahk-till"). These latter two will require small additions to the computation model.

The development environment includes:

1. Visual source editing and browsing
2. Compiling from source to the computation model
3. Build tools, including creating the distribution bundle
4. Team interaction features
5. Debugging tool
6. Test harness

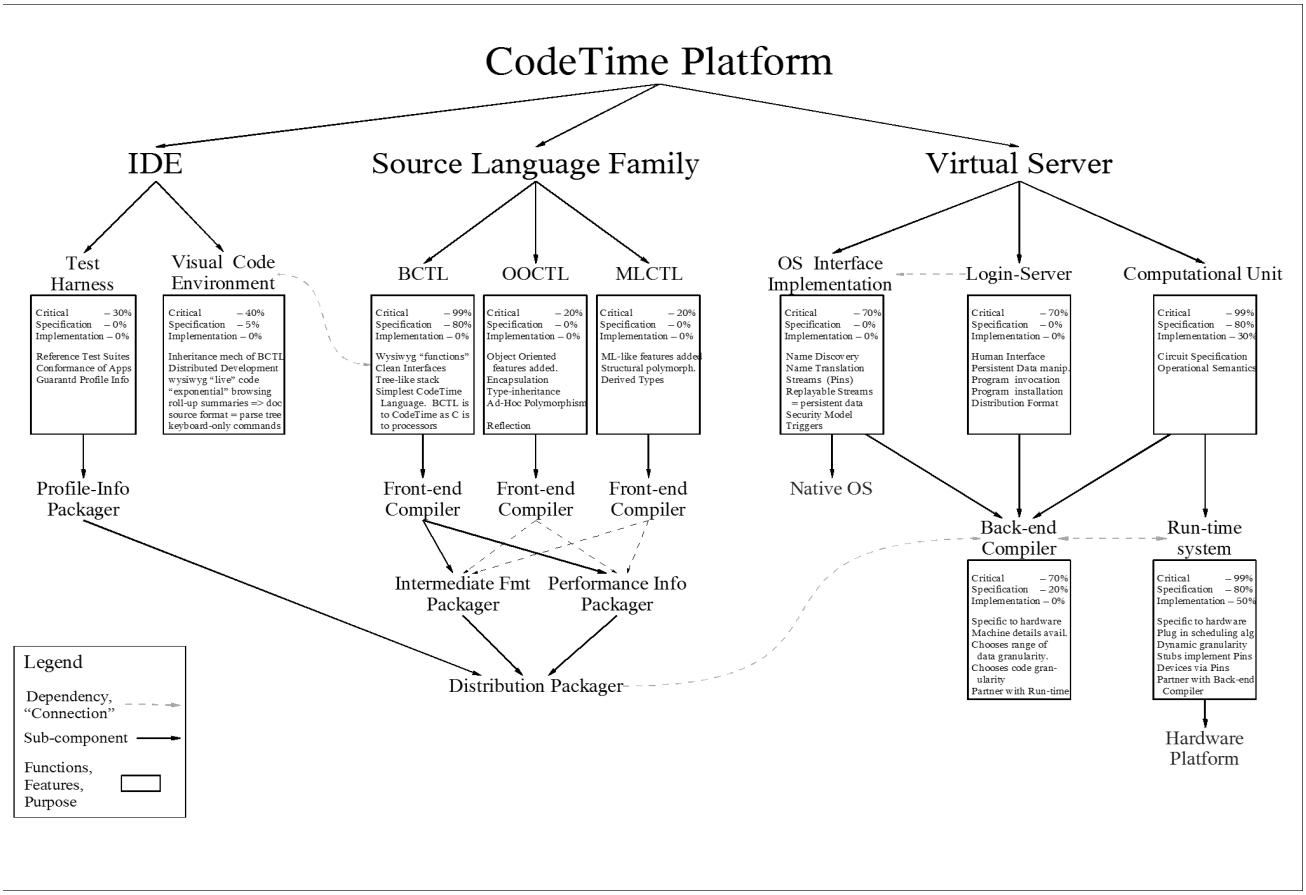

Figure 1.

4 The Virtual Server

A virtual server is defined as three things:

1. A name space
2. The collection of all the systems registered as part of that virtual server (vs external systems)
3. An implementation of the computation model which programs run on

In addition, to be a CodeTime virtual server, it must:

1. implement the CodeTime computation model

2. have each of the default CodeTime systems, each with the default interface and behavior

4.1 The Computation Model

The computation model is circuit based. A CodeTime circuit is composed of units connected by wires. Two kinds of unit exist, system-units and function-units.

A system-unit is a "window" to another system. The system unit is given the name of that system. It has pins to communicate with that system. The pins in the system unit match the interface published by that system.

A function-unit, meanwhile, is defined by the programmer. It contains three primitive elements: a coordination-element, a function-element, and an output-element.

The primitive elements inside the function units perform all computation and coordination of data-movement. The system units connect to systems outside the circuit which perform all the OS functions such as reading and writing data (each file is a separate system), searching for the names of systems, receiving input from a keyboard, displaying data.

4.2 Details of Units

4.2.1 Function Unit Details.

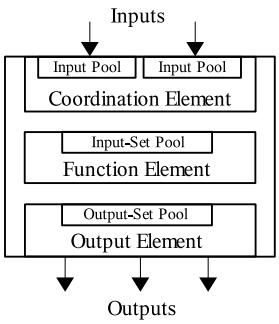

Figure 2. The elements inside a function unit. The coordination element holds the input pools. It creates sets of input parameters from data in these pools. The function element receives parameter sets from the coordination element and spawns an active copy of the function to process each set of parameters. The output element receives a collection of output values from each spawned function-copy and puts those values on the output wires.

The function-unit performs the computation of the circuit. A function unit contains three primitive elements which are wired in series.

Data enters a function-unit by coming off a wire and entering a pool of input-data. These input-data pools are in the coordination-element, which holds one pool for each input. The pools are consumed to generate sets of inputs (called input-sets).

The coordination element chooses candidate input-sets from the input pools, then tests each against a boolean. The program specifies, for each function unit, both the way candidate input-sets are chosen, and the boolean expression.

The boolean looks at values in the containers in the candidate input-set. If a candidate input-set passes the boolean, it goes to the function-element, where it is placed in an "input-set pool".

The function-element takes each set out of the input-set pool, and creates a function-animator for it. The function-animator has a copy of the function and consumes its set of input parameters.

A function-animator is an atomic work-unit, equivalent to a short-lived un-interruptible thread. The process of consuming an input-set is the process of the computation of the program taking place.

Upon completing animation, a function-animator produces a set of output values. It places this set into a pool of output-sets which resides inside the output-element. The output-element consumes the sets and distributes the result-values in each to the input pools of other units.

The order in which things are taken out of the various pools is not specified. Any order is a valid order. In practice, the choice of order will be made by a run-time system. That run-time is created for a specific hardware platform; it will choose an order which is most efficient on that platform.

This is one of the important semantic choices in the CodeTime computation model. As long as the boolean in the coordination element is satisfied for every input-set, the correct answer will be produced. Leaving all other ordering choices free leaves maximum flexibility when implementing on particular hardware.

4.2.2 Systems, Pins, and System Units.

The second kind of unit, the system-unit, is how a circuit interacts with other systems. The structural details are described here, the function and use are described in the subsection of the BaCTiL language which discusses systems and OS functions (5.4).

"System" is the basic unit of "thing" in the CodeTime platform.

A circuit is considered a system.

Each OS function is its own system.

Each data-file is its own system.

Pins are the means by which separate systems are connected to each other. They are how a circuit interacts with anything else.

Pins are used in two ways:

1. loose:: when loose, pins are used by other systems to invoke the circuit and communicate with it
2. grouped:: grouping pins in a system unit is how the circuit invokes another system and communicates with it

Loose pins are declared in the interface a circuit presents.

The interface of a circuit consists of:

1. The number of pins
2. The label and direction of each pin
3. The data-structure that travels on each pin
4. The name of the system

One circuit invokes a second circuit by placing in itself a system unit that adheres to the second circuit's interface.

For example, consider two circuits, circuit A and circuit B. Circuit A has a number of loose pins. It declares these pins in an interface. Circuit B has a system unit in it. The system-unit has the same pins as the interface that circuit A presented. Circuit B presents the name of Circuit A to the system-unit. The system-unit then creates and connects to an active copy of Circuit A.

How the interface is published and checked, and more detail on invoking systems is covered in the subsection of the BaCTiL language which discusses systems and OS functions (5.4).

Stucturally, very little is defined for a system unit. Each system-unit must have:

1. a name input
2. one or more pins, each with a label, a direction, and a data-structure name

Data simply flows in or out of each pin. A pin is treated as a wire. The system on the other side of that pin decides what structure it wants to provide for incoming and outgoing data. However, systems receiving data from a circuit cannot count on any ordering, nor on any time-correlation between data on different pins. Those systems publish the tag data they want and are expected to use that to coordinate data arriving on different pins.

4.3 The Memory model

The memory is split into multiple independent address spaces, called containers. Each container is equivalent to a separate virtual-memory space.

Containers have one or more fields (a field == an address). Each field may hold either a value or the name of another container.

Any container which "travels" on a wire as the top-level container usually has a special field called a tag. A tag holds programmer-specified information. The tag information is only used in the terms of the boolean in a coordination-element.

A container which has a tag is called a coordination container.

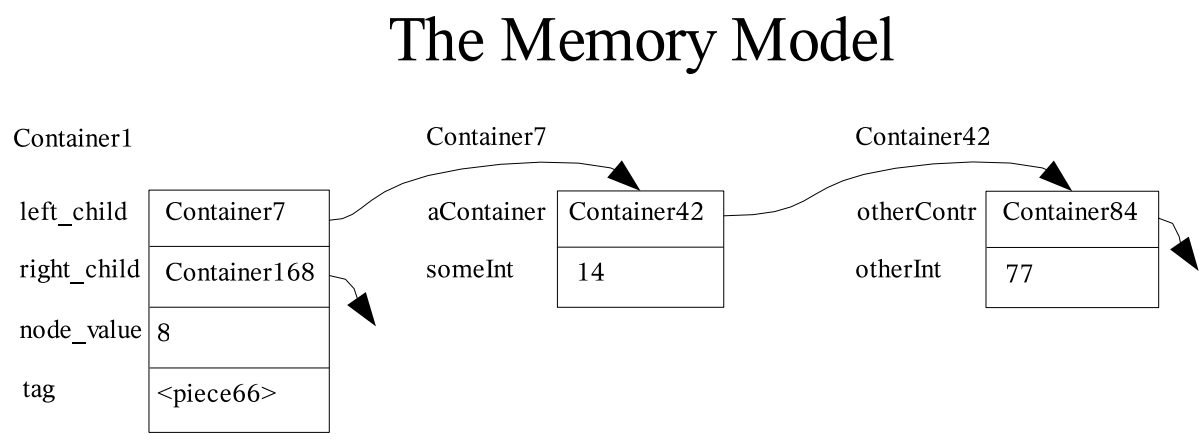

Figure 3. Each box represents a container, which is a separate address space. The labels above the boxes are the names of the address spaces. The labels to the left of each box are the addresses. To the right of each address, inside the box is the memory contents. For example, in the address space named "Container1", the address "left_child" contains the name of another container, "Container7". Container1 has a tag. The tag value is the symbol "piece66". Because Container1 has a tag, it is a special form of container, called a coordination-container.

4.3.1 The World Listener

In order to interact with other systems, the virtual server contains two default systems: a world-connection and a world-listener.

The world listener acts as a server for systems outside the virtual server. It uses the world-connection system to receive requests and send responses from/to those external systems.

The world-connection is implemented natively as part of each virtual server. Typically, it will be a TCP socket listening on a port chosen when the virtual-server is set-up. The implementation looks like a CodeTime system on one side, and a TCP server listening on a port on the other side.

The world-listener, meanwhile, implements the protocol for interacting with a remote machine. External systems which want to be registered in a virtual server's name space must use the world-listener. Humans who don't have the name of a shared-activation on a remote system will have to go through the world-listener of the remote system to ask it to register itself with the virtual server the human's keyboard and display are part of.

The human can also use the world-listener to connect from a non-CodeTime system.

In any case, a human may use the world-server to cause a "shell" program to be run. From the shell they can then perform other interactions such as installing programs, manipulating persistent data, and so on.

The uses of the world-server and detailed examples are covered in the paper on the virtual server[*].

4.3.2

External systems may register

External systems may directly activate a system –

5 The Base CodeTime Language, BaCTiL

The first high-level language for the CodeTime platform is called Base CodeTime Language, BaCTiL (pronounced "back-till").

The language is intended to be authored visually. The platform specifies a visual authoring tool for BaCTiL as part of the development environment.

A program in BaCTiL defines a circuit. When one talks about BaCTiL programs, one uses many of the same terms as when speaking about a CodeTime circuit, such as system unit and function unit. However, BaCTiL programs have heirarchy, whereas CodeTime circuits are flat.

An example of a program's heirarchy can be seen in fig 4. This is how a program will appear in the visual authoring tool.

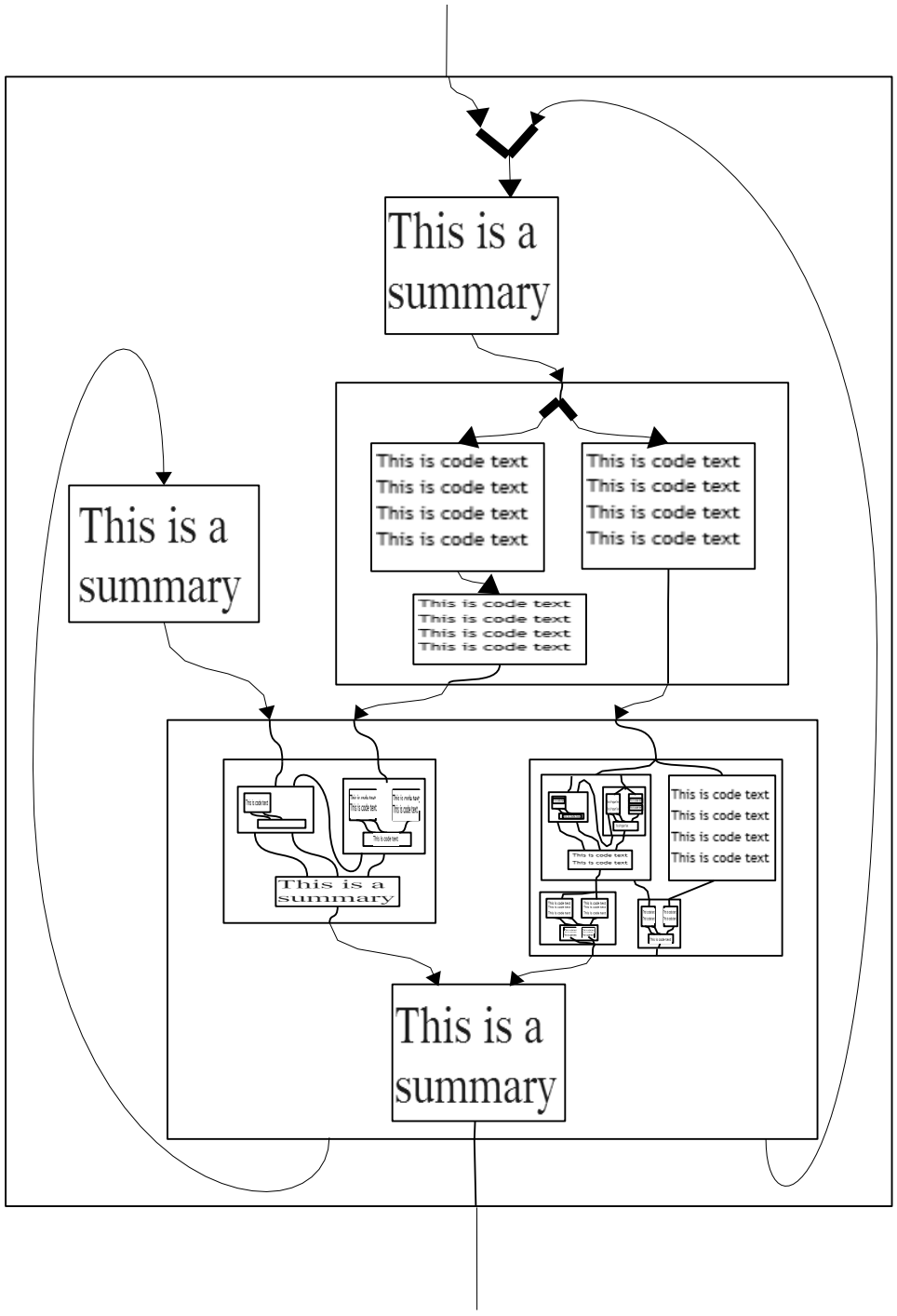

Figure 4. A program written in BaCTiL is a circuit. The circuit is defined heirarchically. The heirarchy can be exposed or "rolled up" and a summary displayed. Just below halfway down in the picture can be seen several levels of heirarchy, with lower levels appearing as smaller boxes inside larger boxes. All boxes within the boundary of a given box are at a lower level. Leaves in the heirarchy are either function units, which contain program code, or system units, which are opaque.

The boxes are pieces of the circuit called code-units. Three kinds of code-unit exist: heirarchy-units, function-units, and system-units.

5.1 Code-Units

All code-units are wired into the circuit by reference. When a code-unit is placed in the circuit, it is really a place-holder that is created. The place-holder keeps state about that reference, including the name of the referenced unit. The programmer sees a rendering of the referenced unit.

These references are live. Each reference appears to the programmer to be right there, and modifyable. When the programmer modifies a unit, three behaviors can happen: they can modify the unit referenced, they can cause a copy to be made and modify that copy, or, if it is a heirarchy unit, they can cause an override-heirarchy-unit to be created and modify that. An override heirarchy unit appears the same as any other heirarchy unit, but it is implemented differently.

When adding a unit to the code, it must be placed inside a heirarchy unit. Thus, the programmer must modify either an existing heirarchy unit, or cause an override-heirarchy-unit to be created from an existing heirarchy unit. The first unit in a program is added to the root heirarchy unit. Every program starts with a default root heirarchy unit.

The add operation creates a reference (in a heirarchy unit). It either creates a reference to an existing code-unit, or it creates a new code-unit (blank or a copy) then creates a reference to that new one.

The programmer must have permission for the operation they wish to perform. Whether the programmer chooses to create a reference to an existing unit, modify a referenced unit, create a new unit (along with a reference to it), or copy a unit (along with creating a reference to the copy), they must have permission for that operation. The authoring tool tracks permissions.

5.2 Heirarchy Units

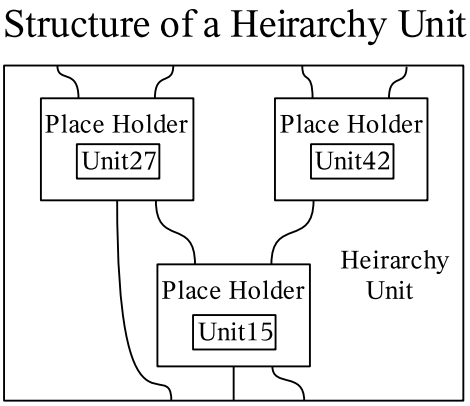

Figure 5. A heirarchy unit contains place holders. Each place holder has a reference to a unit, and stands in for that unit. In the figure, the top-left place holder has a reference to "Unit27". It has a wire connection each place Unit27 does.

Heirarchy-units have no active role in the circuit. They are organisational. They allow easy navigation of the code, encapsulate related code, and enable a form of inheritance.

From the programmer's perspective, when they place and wire a heirarchy unit, they may: create a new heirarchy unit, reference a heirarchy unit, or copy a heirarchy unit. A copy can be shallow or deep. As noted above, all these operations have the effect of creating a reference in the enclosing (parent) heirarchy unit.

The programmer sees the full contents of every heirarchy unit, including lower levels in the heirarchy. They may modify any code they see, if they have permission.

5.2.1 Modifying a Heirarchy Unit

A programmer may choose to modify the internals of any heirarchy unit. They may:

1. change wiring
2. edit code in a function-unit
3. add a unit

4. override a unit

5.2.2 Overridding Code

A command in the authoring tool is used to create a special form of heirarchy unit. This special form has a reference to another heirarchy unit. It looks just like the referenced heirarchy unit. However, any of the place-holders can be replaced. A place holder is made inside the override-heirarchy-unit in the same location as a place holder appears in the original. The replacement doesn't affect the referenced unit.

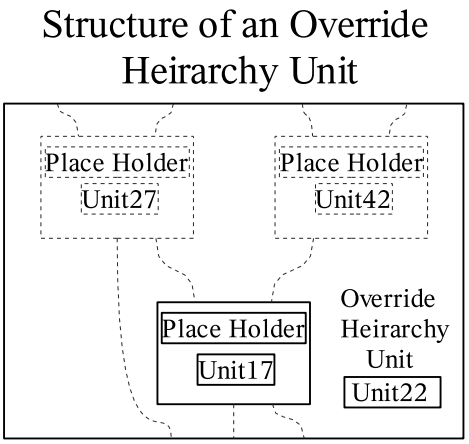

Figure 6. An override-heirarchy-unit references another heirarchy unit. This one references Unit22. It appears identical to Unit22, except for one change. The bottom-most place holder now references Unit17 instead of Unit15. Only the bottom place holder exists in the override heirarchy unit. The other two place holders are in the referenced unit (Unit22).

Thus, when viewing a particular portion of code which has many levels of heirarchy, the programmer may be viewing any combination of normal and override heirarchy units.

When the original, overridden heirarchy unit is modified, an inconsistency may arise between the overridden (referenced) unit and the overriding (referencing) unit. The authoring tool handles this situation. It may either disallow the modification, or it may force changes to all affected overrides.

The authoring tool includes features to highlight which heirarchy units are normal, and which are override-units. It also allows viewing all locations that reference a given unit.

5.3 Function Units

Function-units have two parts: coordination-code and a function.

The function is a straight-line segment of imperative code. It is written in what resembles a simplified form of C. However, it has no persistent data, no loops, and a single invocation may produce multiple output-values on multiple output connections. It has only temporary variables. All data which, in other languages, is normally persistent in the code, such as loop indexes and flags, is bundled with the data operated upon. The entire bundle travels on the wires (as a structure made from one or more containers).

The coordination code specifies how to choose input-sets from the input-pools, and the boolean that candidate sets must pass.

The boolean is an assertion. It relates the values in the input-set elements. Values that are only used in the booleans are stored in tags. This kind of value is a way for the programmer to assign scheduler-relative meaning to data. The program-level scheduler uses tag values to know which data is safe to group together.

For example, inside matrix multiply code, pieces of several different vectors could arrive at a reduction function-unit. This unit must only combine pieces from the same vector. The programmer places a vector-ID in the tag of each piece. The coordination code chooses two pieces from the same input pool. The boolean only passes if both pieces have the same vector-ID in their tags. Those pieces are then passed to the function and added together. The tag would also keep information to track when all the

pieces of a given vector have been accumulated.

```
VectorElem is
 { val: float;
   tag.howManyAccumInThisCell: int;
   tag.howManyInVect: int;
   tag.vectID: symbol;
 }
```

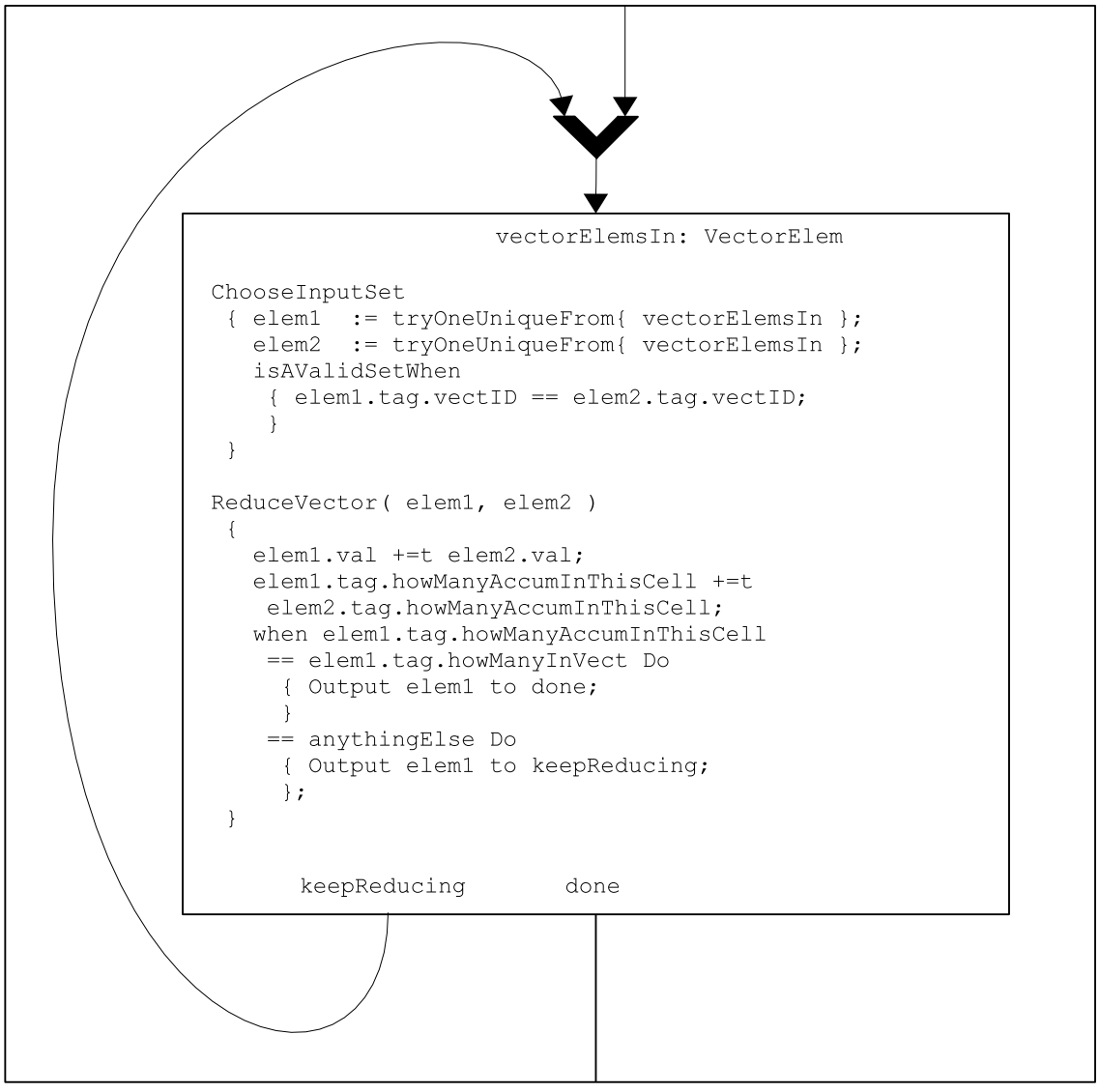

Figure 7. This is actual BaCTiL code. It implements vector reduction. It expects individual elements, of type "VectorElem" to come in from the top. These flow into a combiner (the chevron) which places both inputs onto the same output. Just below the combiner is the function unit. The function unit has two sections. The section which begins with the keyword "ChooseInputSet" is the coordination code. The section below that is the function. The function here is called "ReduceVector" and takes two input parameters. The input parameters are chosen by the coordination code, both from the same input pool (vectorElemsIn), and handed to the function. Every input set spawns a new "thread" which lives just long enough to complete the function applied to that set of parameters. These threads are atomic.

As long as the boolean statement is satisfied, the cells may be combined in whatever order, in chunks of whatever size, and the result will always be correct. In the sample code in fig 7, there is no telling how many elements will build up in the input pool, nor which elements will be added to which. All that is known is that only elements from the same vector will be added together.

5.4 Systems and OS functions

A circuit is most useful when it can interact with things outside itself. Typical useful things include keyboards, displays, persistent data, and web-servers. Each is treated as an independent system.

A system that a program-circuit connects to is either internal to the same virtual server, or external. Examples of internal systems:

1. Persistent Data After being installed, each file is treated as its own system
2. Installed Programs After being installed, each program is treated as a system
3. OS functions Naming system, real time clock, and so on, are each a system
4. Authentication Each virtual server implements security in its own way, as a system

Systems in different virtual servers and programs running in non-CodeTime environments are external systems. Web-servers and databases are good examples of external systems.

5.4.1 The OS Interface

All operating system services that a program needs are implemented as systems. The set of systems implementing these functions is known, collectively, as the OS interface. It will likely be common to implement these systems by a translation layer that in turn calls a native OS, such as Linux.

The standard systems that every virtual server should implement:

1. Name system (used to install new program-circuits and search for name of installed systems)
2. Authentication system (used to obtain security credentials and query permission for credentials)
3. Standard files (used to find out information about the virtual server)
4. Standard Input and Output system (keyboard in, text stream out)
5. Standard GUI system (a graphical display system like SDL, plus input device like mouse)
6. Real time clock system (used to discover current time and to schedule notifications)
7. The Standard Server (listens to outside world, allows an external system to initiate interaction)

5.4.2 Activations

Strictly speaking, a circuit is a blue-print. When a circuit is run, an activation of that circuit is created. It is that activation that performs work.

There are two kinds of system: multiple-activation systems, and single-activation, or singleton systems.

Many hardware systems are singletons. A CD-ROM drive might be a singleton, for example. The standard server is defined to be a singleton.

A circuit-activation is created by presenting the name of the circuit to a system-unit. The system-unit either causes a new activation to be created and connected to, or it connects to an existing activation (for singletons).

External systems can contact the standard server and request that a circuit-activation be created and connected to the external system (or else that an existing activation be connected to the external system).

Every system must be installed into the virtual server before a circuit can initiate connection to that system. (A generic web-server system is standard, and generic servers of other protocols may be added.. create an activation of the generic thing, then give it the "connect" command containing the URL.. the generic system has the protocol for the external system built into its adaptors. The generic system implements higher-level adaptors which turn, for example, HTTP request objects into command-structures (? better examples?)

Installing a system means giving the interface to that system.

Part of installing is stating a contract. The contract says how many pins the system uses to interact, it says the name that a circuit should give each pin, and it says the data-structure that goes into or comes out of each pin (one data-structure per pin).

In addition, the interface has an adaptor for each pin. An adaptor turns a bit-stream into CodeTime structures or vice-versa.

Standard Systems, like the OS systems, which are part of the virtual server, are "installed" by hand as part of the process of creating the virtual server.

5.4.3 Recursive Circuit Activation

A circuit may include a system unit which creates another activation of itself. Restrictions exist on when this is allowed. The presentation of the name to the system unit must have a conditional clause. The front-end compiler must be able to determine that some combination of inputs exists which will cause the clause to not present the name, thereby preventing activation, thereby ending the chain of recursive activations.

Recursive circuits are discouraged. This form interferes with performance. It prevents both the front-end and back-end compiler from effectively optimizing the circuit, and from effectively tuning granularity of parallelism to the hardware.

Recursion can be accomplished within a circuit by using special tags and a particular circuit pattern. This method does not have any performance drawbacks.

Recursion can also be turned into iteration for any finite recursion depth.

5.4.4 Adaptors

External systems which have their own format for bits and bytes must have some way of transforming that data into structures made up of containers.

This is accomplished by adaptors. Any system external to the virtual server must have an adaptor on each pin that system has.

A system must register with the virtual-server's naming-service before it can be connected to by any circuits in the virtual-server. Part of this registration is stating a contract. The contract says how many pins the system uses to interact, it says the name that a circuit should give each pin, and it says the data-structure that goes into or comes out of that pin.

In addition, it gives an adaptor for each pin. The adaptor translates between bit-stream and structures of containers.

The virtual server provides special key-words for adaptors. These key-words can only be used inside adaptor circuits. One set of keywords allow an adaptor to turn arrays into containers, and vice-versa. Another set (plus interactions with OS systems) allows the adaptor to open one or more connections to/from the external system.

The virtual server has open-connection, send on, and listen on keywords for adaptors. The communication protocol is implemented as part of the virtual server implementation. A virtual server may only communicate with external systems that use a protocol the virtual server has implemented. Examples of protocols include TCP/IP over ethernet, SCSI, fire-wire, etc.

From there, the adaptor looks at the bit-stream, knows the bit-format, puts those bits into the appropriate arrays, converts those arrays into containers. It then uses the special keywords to cause the names of containers to be inserted (names can never be seen explicitly in a circuit, the adaptor must ask the virtual-server to insert names for it). When finished with a data-structure, the adaptor uses another special key-word to cause it to appear in the circuit.

The adaptor must also implement all system-specific communication-protocol with the external system. However, control protocol which has program-level meaning is handled via command and response pins. The adaptor must take from the circuit, the container which has command information and translate it into the correct command-message format for the external system, and vice-versa.

5.4.5 System Units

System units are used to interact with other systems. They collect all the pins for interacting with that system into one unit.

A system-unit has two parts: an input which accepts the name of the system being interacted with, and a number of pins to perform that interaction.

A system-unit is placed in the program (circuit) where the programmer wishes to wire interactions with that system. The pins in the system-unit must be labelled according to a contract that the system connecting-to published when it was registered with the naming service. Typically, some pins will be used to send commands to the system, others will be used to send and receive data.

Each operating system function is treated as a separate system. Each file is treated as a separate system. All of the standard operating system functions, such as a name-discovery service, real-time clock, standard console input and output, and so on, are defined for the CodeTime platform. Each is a separate system which is implemented as part of the virtual server implementation. Each has a standard name which is used to connect to it.

Persistent data is installed into the virtual server. It arrives in a data-bundle from an external system, such as a CD-Rom. The bundle contains:

1. the name of each file in the bundle
2. the contract for what pins connect to the file, with the data-structure on each pin
3. an adaptor to translate between raw bits in that file and the data-structures
4. meta-data about the file, used by the name-discovery system (the search service)

Once a file is installed into a virtual-server, it is a named-system just like every other system in the virtual server.

Different kinds of files exist. For example, some files are searchable. One may send a command to the file to give all structures in it which pass a certain criteria. Other files are installed read-only, with no search. These files take only a command to start streaming from the beginning, and to stop streaming.

The figure (fig 8) shows a system unit for connecting to a typical operating system function. For instance, this system-unit may be used for connecting to a data-file that has been installed into the virtual server.

The name of the system (eg file-name) is sent in a container to the “Name of System” input. The container has just one field. It contains a symbol, which is the name of the system.

Then, commands and data are sent to the system on the wires connected to the pins at the top, which go from the circuit to the system.

For example, files may be installed into the virtual server in searchable form. If the system unit in the figure were used to connect to such a file, then commands would be sent to the “Command To” pin.

Each command has a command-ID tag. The command-responses and the data-from both carry the command-ID that caused them.

So, all structures in the file which pass the search criteria would stream out of the “Data From” pin. Each of these structures come out in a standard carrier container. The carrier-container has the command-ID in its tag.

It is possible that multiple system units in the same circuit try to connect to the same name. Three cases may arise for multiple system-units all connecting to the same system (the same name):

1. multiple connections are disallowed
2. more than one system-unit are considered the same connection
3. each system unit is considered a fresh, independent, connection to that system

The nature of the system being connected to determines which case arises. The standard systems (the operating system functions) all have standard behavior defined for multiple connections.

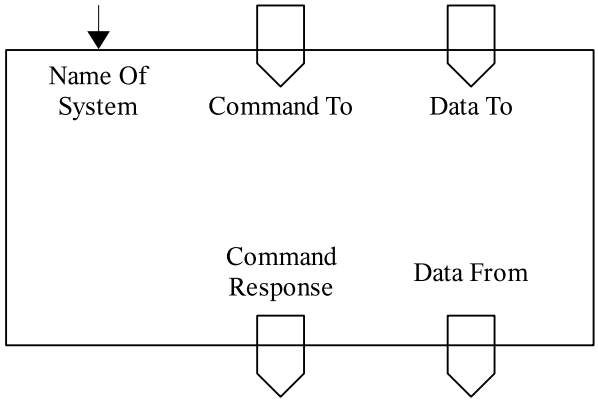

Figure 8. This is a system unit. It has an input at the top left which is sent the name of the system to connect to. The pins at the top send data to that system. The pins at the bottom stream data from the system into the circuit. The lables on the pins in this system unit are the standard labels used for connecting to operating-system functions.

5.4.6 Security

Each system has its own security criteria. Some systems, such as the real-time clock, have no security checks. Others, such as a file, do have security.

For systems which require security clearance, an authentication is included with each command. An authentication is a special container which is created by the virtual server's authentication system. It carries all the information used to determine permissions. An authentication cannot be looked at by a circuit. Only its name can be operated upon, never its contents.

Authentications are tracked by the virtual server. Most circuits require an authentication in order to activate them. That activation of that circuit then has that authentication available to it, should it choose to use it. The circuit may also opt to use the authentication system to create a new authentication. This would involve the authentication service asking for a password, checking a digital signature, etc. Each activation of a circuit is separate from all other activations. Authentications which are created as a result of a particular activation cannot get into other activations, except via pins. Pins perform permission checks when an authentication is sent over them.

It is important to note that these semantics leave a lot of room for implementations to vary.

For example, consider a program which streams extremely large matrices from two files into a multiply circuit. From the perspective of the program, the data streams are started, and flow into some function unit. The function unit code is written to expect matrices composed of normal arrays. However, from the run-time implementation point of view, if the arrays are very large, this is impractical. They may not even fit in a single processor's address space.

One solution is to write the run-time system to convert all incoming persistent data. Array containers which are larger than a certain size are stored in two parts. The first part has the meta-data about the second part, which has the actual data. The conversion happens when the data comes from outside the virtual server. For example, if the data comes in

An example run-time system is the tree-of-graphs peer-to-peer one detailed in a companion paper[*].

The back-end compiler is written in conjunction with this run-time system. It compiles two kinds of executable for any units which take arrays in their inputs. One produced executable module is for leaf-level peers, the other is for higher-level peers.

The module for higher-level peers tests for, then operates on the meta-data. It doesn't perform any of the actual computations. It only performs division and un-division.

The leaf-level executable module is a normal module. It performs computations. It operates on normal arrays. To make this work, the run-time on leaf-level peers performs the conversion from meta-data to actual data. It uses the meta-information to fetch a portion of the original, large, array off the disk, or another leaf-level run-time. This data is given, as a normal array, to the executable module.

The group will pass around actual data among themselves. When done, the leaf-level run-time checks the size of arrays in the result. If large, it re-converts to meta-data + actual data. It then hands the meta-data to its parent. The actual data will remain on the processor which produced it until some other leaf-level run-time somewhere requests it.

6 The Development Environment

The IDE implements the visual behaviors of the BCTL language. It also enforces testing within a defined test-harness which automatically collects profile data and packages that data with all intermediate-format code. In addition, the IDE is encouraged to provide a number of useful kinds of information with the intermediate-format code, including means for early detection of array bounds violations, transforms for tag-code which allows efficient implementation, etc.

A visual orientation was chosen because it felt natural to express a circuit that way. This means the code is wysiwyg (what you see is what you get). Where a traditional language would have a function call whose definition must be searched for, Code Time has the function-call code immediately in view, just zoom in. Further, the flow of data is the same as the flow of control, so looking at the code shows the control-flow visually. These features localise the code which must be considered while programming, and thereby provide a modicum of self-documentation. These features also enforce clean, anonymous, interfaces, which enhances code reuse.

How it encodes language semantics

how it provides performance information

how enforced test-harness ensures profile info which is important to performance.

how debugging is easier (due to analysis tools runnable on code to detect race conditions, and ability to single-step, and ability to trace the pieces of data the data-in-error was combined with. That can simply be checked backwards until the incorrect combination is found. The data-trail equals the control-path trail, and is easy to trace and analyze later.)

The well-specified virtual machine, strong type system, lack of explicit synchronizations, and data flow style combine to make debugging straight forward. Break-points can be set on tag-values, code can be single-stepped through data productions, and ensuing results on wires browsed in the original visual environment used to develop the code.

The performance of each function is straight forward to determine due to the imperative semantics and simple syntax.

The amount of parallelism is apparent from considering how many simultaneous elements each function can produce and how many simultaneous functions may overlap in consuming data. The coordination language encodes this behavior in a way which yields easily to casual analysis.

different implementation strategies can be quickly compared for which has better potential performance.

The net effect is the ability to write and compile a program once, install the result on a wide variety of parallel machines, and obtain high levels of performance on all of them. This in a language easy to write, easy to debug, highly reusable, self-documenting, and amenable to reasoning about relative performance.

Part of the client API allows for an application-launching client to be installed on a workstation. In this case, a user may, at their discretion, keep data-files containing their work on their local machine or on a network file server. The file extension used by the CodeTime application which created those files is associated to the application-launching client. This can be done with a mini install-script which, when run on the workstation, interacts with the application-client to tell it how to launch the CodeTime application. The install-script also tells the application-client to associate the CodeTime app's extension to the launch-instructions. The install-script then goes on to tell Windows, for example, to associate the CodeTime app's extension to the application-client.

After this install is complete, a user can double-click on a file, windows will launch the application-client which in turn will connect to the CodeTime virtual machine, via the local stub, and then run the CodeTime application, passing it the name (including local workstation name) of the clicked-on file. If the virtual machine is local, or not currently up, the app-client first launches it, then connects.

The stub plays a dual role. It acts as the single point of contact between the virtual machine and an end-user's workstation. Thus it not only implements the client API, but also acts as the means for data exchange between the CodeTime system and the local workstation. In particular, user interaction coded in a CodeTime program passes its communication through the stub. The application-client may be configured to act as the standard line-based input and output. Or, if graphical interaction is desired, then a local-workstation-specific program must be written which uses standard stub-calls for data-exchange and workstation-specific graphics code for display. Technologies such as Java's Swing may be used to make the graphics portable.

On the local-side a standard API enforces a machine-independent interface. On the CodeTime side, the system administrator configures the stub to connect in whatever specific ways were chosen for that installation. This could be via TCP to a URL, custom high-performance network code, or, in the case that the entire CodeTime system is local, the stub could simply be a DLL that implements the entire virtual machine. This allows application developers to write graphical code for only the 3 standard platforms (Windows, Mac, XWindows), much as they do now. Meanwhile, they develop only a single version of the computational code which will run across all CodeTime enabled parallel hardware, much as Java code currently does for serial machines and, in limited ways (like App-servers), for parallel hardware.

Some of the functions available in the user-facing client include browsing for Code Time applications to run, looking at performance statistics from previous runs, discovering the characteristics of the virtual machine such as configuration, computational power, etc, and any special productivity enhancing features that the client writer chose to implement, such as automating a set of runs while varying inputs.

6.1 The Developer Environment

The developer environment is a system of inter-connected GUI tools which include (fig 9):

- code browser
- editor
- debugging visualizer and command interface
- bug-tracking
- test harness in which debugging and test activity takes place

Figure 9.

6.1.1 Code Browser and Editor

The code browser is a defined part of the language. The intent is for the language to be developed in a graphical manner. This enables behaviors not possible in a text-only format. However, an Architecture Definition Language does exist for pure text entry of code. The browser integrates with the editor, popping up the code for whatever box is being viewed.

To reduce the extraneous work introduced by managing creation and placement of graphical elements such as boxes and wires, a set of keyboard commands is included as part of the CodeTime specification. Keyboard-only (no mouse) commands exist which allow creation of new boxes, creation of wires between boxes, moving focus from box to box, and each, other, code-manipulation operation, without touching the mouse. The exact key sequences are, of course, customizable, and extensions are as well possible. To implement these, VLSI CAD place-and-route techniques have been suggested for placement and wiring of boxes.

6.1.2 Debugging Visualizer

The debugging visualizer allows single-stepping through data-arrivals at a given box, setting breakpoints with conditions of arrival of particular data at particular points in the data-flow graph, and specifying logging points. Bugs logged into the bug tracker usually have a set of data which was captured by the test harness. These bugs may be brought up in the debugger and the associated data viewed and stepped through. Data is viewed by seeing it appear inside the code browser. Any wire may be looked at to see what data is on it at any given break point. Further, the execution of the internals of a particular box may be single stepped through.

This style of debugging allows logical errors as well as errors in handling side-effects to be easily spotted in a systematic way. The location of the error can be tracked down by determining which box the error occurs in and at which point in the data-flow. To help ensure that all side-effect errors show up, the test harness runs all code with a highly random ordering. That is, the scheduler randomly chooses start times, from the set of valid ones, for each box instance. Thus, if any race conditions or deadlocks exist in the test data-set, they are likely to exhibit themselves. Further, theoretical work suggests most, if not all, potential race conditions and deadlocks will be detectable at front-end compile time.

Once erroneous data is detected, its ancestry may be traced back up the data stream to determine which box the error occurred in, or at which point a side-effect caused the error. Thus, the particular sequence which caused the error does not matter, only the presence and ancestry are needed to isolate the origin. Once the error producing box is determined, the error is either a logical error in the box, or a side-effect error. Side effect errors occur when the constraints on side-effects are under-specified, and fixed by adding constraints.

6.1.3 Test Harness and Bug Tracking

The test harness is used to run all debug sessions, including unit tests by the developer, manual tests by QA, and automated tests. Any bugs are automatically logged in the bug tracker, along with exact versions of code, exact machines, exact compiler versions, and a set of data collected during the run.

The test harness also collects profile information on every run. This is automatically logged in the development-environment's profile database. The front-end compiler accesses this data base on every compile of the code.

6.1.4 Front-End Compiler

The front end compiler is the one that the developer uses. It checks syntax, does type checking, and performs operations which generate special code for use by the back-end compiler and virtual machine. It uses the profile information supplied by the test harness, and also places that profile information into the intermediate format.

The output from the front-end compiler is the distributed format of the code. Once the front-end has completed, it has produced an intermediate format which both represents the code structure and communicates meta-information for use by the back-end compiler. This is placed upon CD ROM and/or downloaded over the Internet as the "executable."

6.2 The Forms of Code During an Application's Lifetime

This section provides both a 10,000 foot view of the syntactic elements of the source code, and states the various forms the code gets translated into over an application's lifetime (fig 10). Many terms will be used with only brief explanation. However, fuller treatment of the terms is presented in later sections.

An application in Code Time has three main embodiments:

- source code
- partially-compiled intermediate code
- executable machine code

Figure 10.

6.2.1 Source Code Formats

The developer enters the source code, usually via a GUI. The development environment tools translate from the GUI format into a text-only form (which the programmer may choose to enter directly). The front-end compiler takes as input this text-only form and produces a "compiled" intermediate form (fig 10).

The elements in the GUI format (fig 1) are boxes, wires, splitters & combiners, and text. The kind of box may be:

- "leaf" (or equivalently "code") box, which contains the text of (mostly) imperative-style code
- "container" box, which contains data elements, of any type
- structure definition box, which defines, in a C-like struct style, data types
- system box, which implements a generic form of an OS function

A wire shows the flow of data from one box to another. Data flows only one direction on a wire, leaving from the bottom of a box and entering at the top of a box. A splitter copies its input to its two outputs. A combiner places both its inputs onto the same output wire. An inputs is always at the top (of the box, combiner, or splitter), an output is always at the bottom.

The text form of the code for boxes and wires has two sections. The first section defines the types of boxes, and the second section declares instances of boxes and interconnections between them.

A box-type definition includes:

- a declaration of input types
- a declaration of what qualifies as a valid set of input data
- constraints on the ordering of executions of that type of box, based upon tag values
- imperative code
- side-effect constraint code
- tag modification code

The box interconnection code includes:

- specific instantiations of boxes, including specification of each instance's type
- instantiations of combiners and splitters
- wire statements which connect specific outputs to specific inputs. Inside container boxes, wire statements may alternatively connect box inputs to container inputs

6.2.2 Compiled Intermediate Format

The intermediate format of application code (fig intFormat) is the form distributed by application developers. The information in this format is split into two categories:

- a graph-based representation of the code

– additional information attached to the graph structures, such as profile info, identification of loops, and variable lifetimes and type information. This info is for use by the back-end compiler, scheduler, and run-times inside the VM.

The graph representation (fig graphRep) has a node for each box instance, and an edge for each wire. A box node contains another graph if it is a container box, or an array of operation trees if it is a leaf box. An operation tree has nodes which represent primitive operations such as arithmetic, comparison, branch, assignment, association following. Leaves represent values to be operated upon, either literals or final-association-links which yield the values. Each loop is given an identifier and all boxes within that loop marked with the loop identifier. Markings show places where transport strucs are created, where they become undefined, where side-effects are introduced, and which operators are under side-effect protection. Side-effect constraint code is attached to boxes and operators as appropriate. Finally, pairing code which declares what constitutes a valid input set, ordering rules for the inputs, as well as type information on inputs and outputs is attached to each box-node.

The additional, attached, information is encoded in a standard format. This format may change over time, but only in an additive way. That is, new kinds of information may be added, but the shape and presence of information specified in older formats must always be included. This allows an extensible format that works in all combinations of new and old front-end and back-end compilers.

Other kinds of additional information includes:

– profile information specifying the probability taken for each branch, and normalized frequency for each box

– mini-programs generated by the front-end compiler which are represented as arrays of operator trees, just like the regular application code. These mini-programs are for special-purpose calculations, including these:

  → Expected number of "canonical instructions" executed by a given type of box, as a f() of data-size.

  → The number of iterations a particular loop will execute, given a value on a wire at a particular location in the graph (IE, a mini-program which takes a set of values from a particular set of wires and produces the number of iterations a given loop will perform).

  → The maximum array index computable at a given array-access line of code. This mini-program takes as input a particular set of values on a particular set of wires.

6.2.3 Installed Machine Format

The intermediate format is distributed by the developer. It is the form received by an end-user. During the installation processes, this intermediate format is compiled down into a number of hardware-specific forms. The virtual machine handles installations and invokes the appropriate back-end compilers. The VM also tracks each machine-specific form in some type of database (centralized or distributed). It tracks the exact make, model, and version of the back-end compiler used to generate the code, and of the hardware for which the code was generated.

Thus, each machine requires its own Code Time back-end compiler, with specific optimizations and code generation tuned to that exact machine. This contrasts to current practice, in which, due to wide distribution, code is generated only for a generic machine. In Code Time, however, code can be generated which takes advantage of exact knowledge of the specific hardware, including cache size, memory hierarchy parameters, chipset, peripheral bus type, etc. This is one of the places where the profile information contained in the intermediate format is used. For example, on a single processor machine, an entire loop may be compiled as one block of code, including unrolling it. Specific optimizations can be performed such as eliminating tags and instead directly passing a pointer to transport strucs from function to function.

In some virtual machines, the executable binaries may actually be modified during a run. That is, dynamic optimizations which modify the binary according to run-time statistics may be implemented under the covers. Also, future language features such as introspection and various forms of polymorphism may cause run-time updates to the machine-format of the application.

The effort expended in machine-specific compiler optimizations is, again, amortized across every application run on the hardware. All the applications, past, present, and future, benefit from each one-time machine specific compiler effort.

6.2.4 Pins

Pins are a form of system box used to input-from and output-to other systems. For example, any user interaction taking place in a program is performed via pins communicating with a user-interaction application running on the machine the user is engaged with. This could be an adapter to some third-part program like OpenOffice Calc, a GUI front-end running on the user's workstation, or a server running JSPs which produce HTML that the user interacts with via a web-browser. For example, in the HTML case, the Code Time application would receive requests from the JSP server, via a stub on the JSP machine. These requests would enter the Code Time application by becoming transport structures flowing out of an input pin. The Code Time application would then perform computation and return the resulting data to the JSP server by placing a transport structure on a wire connected to an output pin.

An adapter must be written, which runs on the external machine, that goes between serialized transport structures and the external program's required format. On one side, an adapter interacts with the Code Time stub resident on that external machine. The adapter receives serialized transport structures from the stub, and gives serialized transport structures back to it. On the other side, the adapter gives and receives whatever format the external program requires. If the external program is a GUI written specifically for the particular Code Time application, then it will understand serialized transport structures directly, and no adapter is required. By placing the adapter on the external machine, the Code Time code remains free of any kind of machine detail.

6.2.5 Divider and UnDivider

The Divider and UnDivider are special application-programmer-coded boxes which enable hardware-specific, yet hardware-independent data-parallelism (fig Divider). They act as agents that interact with the virtual machine's scheduler. The scheduler may decide, based upon its internal implementation and internal knowledge of the machine structure and current state, that a particular piece of data needs to be divided into several pieces.

At this point, the scheduler begins a negotiation with the Divider. It generates a preferred number of equal-size pieces, then hands the Divider the data plus preferred number of pieces. The Divider responds with a structure indicating the number and sizes of pieces it could actually produce. If this response is acceptable, then the scheduler sends the structure back and tells the Divider to make it so.

The Divider then produces the pieces, either by copying the original, or by assigning pieces of the original to different structures via side-effects. The thus-produced pieces are handed back to the scheduler. The scheduler then proceeds to output the pieces, one by one, to a Body box. Physically this may involve moving the data around the machine and sending execution-instance bundles to leaf-run-times.

The Body box is coded to perform the main computation on any size piece that the Divider is coded to produce. The Divider has provided all the information about boundary-conditions, size, etc, that the Body box will need.

The UnDivider box then receives the results from the Body box one at a time. It uses the tags in the transport-containers carrying the result data, plus information sent by the Divider, to aggregate the results into the final, overall result. The UnDivider is also written by the application programmer. Of course, the undivider is free to, in turn, use its own divider-undivider pair internally to help parallelize the task of putting the original pieces back together. Likewise, the Body box may also have its own embedded divider-undivider pairs.

The Divider-UnDivider pair allows complex data-structures to be efficiently divided across heterogeneous hardware. It represents a clean interface between the hardware-aware virtual machine and the data-structure-aware program. The programmer only has to know how to divide the data up into pieces. The scheduler only has to know how to decide the size of a piece. Together, the same compiled code can run efficiently on a grid-computer made up of thousands of different machines, or a dual-threaded PC (see Matrix Multiply sample program).

7 Conclusion