BaCTiL: Base CodeTime Language

by Sean Halle

Email: seanhalle@yahoo.com

Abstract

This paper describes BaCTiL, a high-level language for the CodeTime computation model. It is a simple language: BaCTiL is to the CodeTime computation model as C is to the Von Neumann computation model.

BaCTiL is best viewed as a collection of three inter-related languages: a visual circuit description language, a coordination language, and an imperative language. A program in BaCTiL is a circuit, composed of a number of units wired together. Each unit is either a system unit, equivalent to an OS system call, or a function unit. Function units are filled with imperative code. The invocation of the function units, with respect to which data and which order, is controlled by a coordination language.

1 Introduction

The CodeTime platform for parallel software aims to enable write once, compile once, run anywhere for parallel software. The platform consists of a virtual server, which embodies the compution model; a family of languages which compile down to that computation model; and a platform-defined development environment, which enables conformance enforcement and guarantees performance information such as profile information gathered during testing.

BaCTiL is a simple high-level language which is designed to easily compile down to the CodeTime computation model.

2 Sample Code

This part of the paper can be safely skipped. It is included, first, for readers who wish to get their feet wet and see what a full program looks like before proceeding on to the more exact discussion of the language.

2.1 Hello World

The Hello World program (fig 1) shows a simple BaCTiL program. This graphical format is the preferred form for developing code. The graphic-centric approach provides a level of self-documentation, and enables tool features only possible in the visual format which, in turn, enhances code reuse.

The graphical elements seen here are: ellipses which represent system functions, rectangular boxes which contain user code, a trapezoid which indicates a data-stream, wires which represent data flowing, and a chevron which represents the combining of data from two wires onto one.

The algorithm starts with a string, generated in an initialization box, and then loops through the string one character at a time, outputting each character to stdout. This program flow can be understood by inspection of the wiring. The main loop is evident, as well as the flow of data to the stdout system ellipse.

Execution begins at the "start" ellipse. Ellipses indicate "system" functions which are typically carried out by the underlying OS. In this case, the initiation of execution happens when the start box (the terms box and ellipse are used interchangeably) emits a transport container with the `<StartToken>` symbol inside it (the < and > delimit symbols). This in turn triggers the first box to create an instance of the transport structure which is used in the main loop of the program.

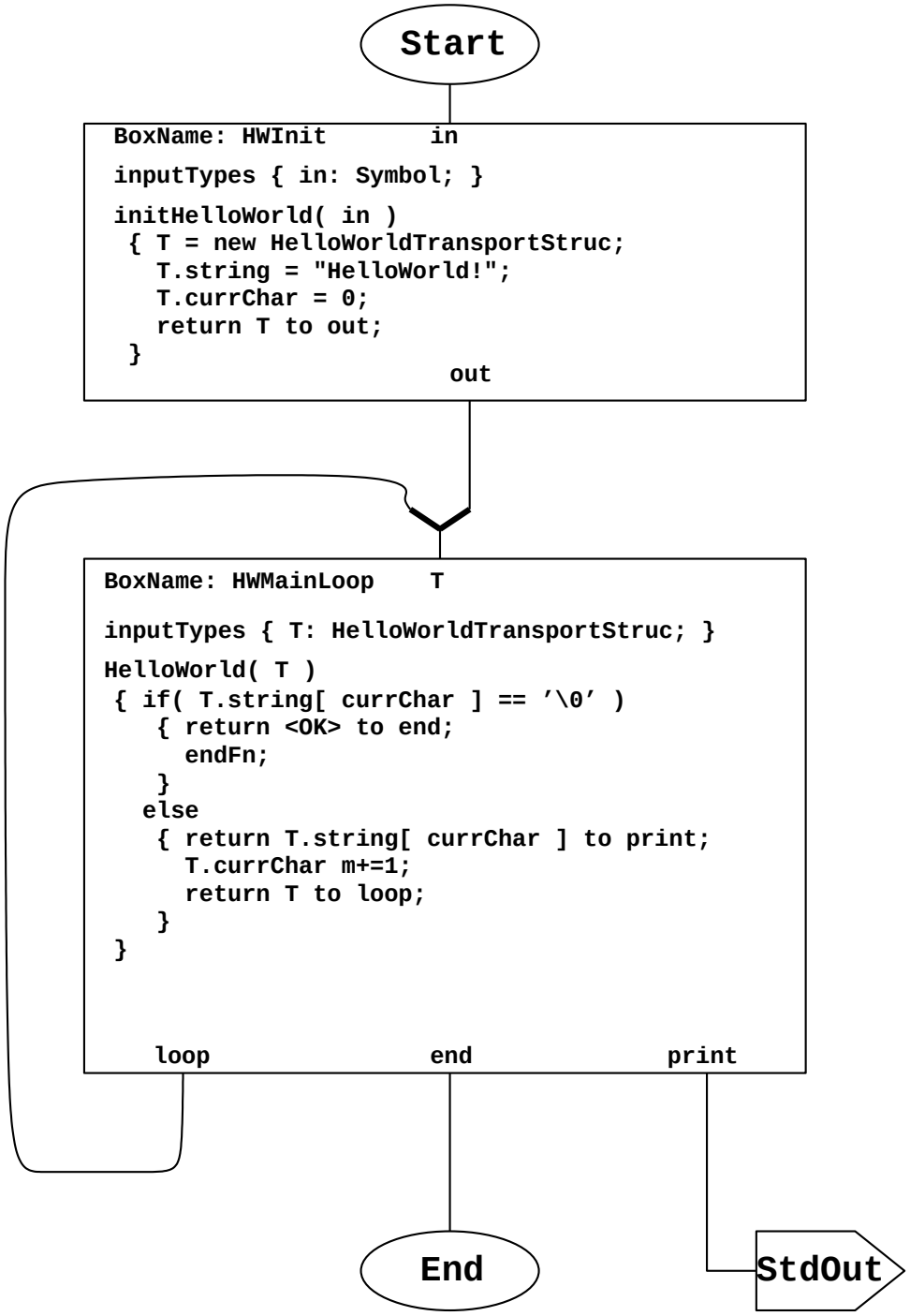

Figure 1.

This is the Hello World program written in BaCTiL. The ellipses are system boxes, which embody OS-like behaviors. The trapezoid is a special form which represents a stream of data flowing out of the system. Inside the rectangles, the functions execute each time data arrives at the input (at the top). Data travels, held inside data-containers, on the wires between boxes.

The `Start` box (ellipse) begins execution by placing a symbol-container on the wire coming out of it. When the container reaches the box below, named `HWInit`, that arrival triggers execution of the `initHelloWorld` function. This function creates a new transport-container and sends it to the next box, `HWMainLoop`, which loops through the string, sending each character to the `StdOut` data stream. When done, it sends a symbol to the `End` system box to signal to the system that execution is complete.

Transport structures (or more correctly instances of them, called transport-containers) are the vehicle by which data gets carried along the wires. All data in code time, semantically, lives in a "data-container." Variables and wires hold associations to these containers. However, when a container is placed onto a wire, it must have a tag in it. The presence of a tag distinguishes a transport container from other kinds of data-containers. The purpose of tags is to coordinate data. Tags are not shown in the hello world example; they will be covered in the next program, QuickSort. The important concepts here are that:

1. all data lives inside a data-container
2. data-containers placed onto wires have tags in them and are called transport containers
3. structures are often defined, like the `HelloWorldTransportStruc`, which are used to hold program state. For example, the `currChar` member holds the loop variable.
4. transport-containers are usually instances of transport-structures. They typically hold code-related data, while work-containers hold data which is more closely related to final-result data (see Part III the data-model).

The distinction between code-data and work-data is a matter of convenience and may be confusing at first. The terms are used to help explain programs and have no mechanistic implications upon the execution of a program.

The box which receives the start token and makes the transport structure, HWInit, has fairly simple code in it. It uses the "`new`" keyword to make a new transport-container, then initializes variables in it, and returns it to the box's output. Boxes whose only function is to make a transport structure for use by the rest of the program are colloquially referred to as initialization-boxes. This term, also, has no precise meaning in the language and is used only when explaining a program.

A code-box contains inputs (at the top), outputs (at the bottom), a declaration of the type of each input (and optionally output types), an optional declaration of what constitutes a valid grouping of data from the inputs (this is a major use of tags), a function which lists the steps of the procedure the box carries out, optional side-effect specifications which make side-effects safe (another use of tags), and optional tag-generation code which declares what the tag of out-going data should be. More on the structure and syntax of boxes in Part IV Architectural Description Language.

The transport structure flows out of the initialization box into a chevron. This chevron is a combiner that takes data from both inputs and places them on the output (at the bottom).

Inputs to boxes are always at the top, outputs at the bottom. Thus, the inputs to the chevron are at the top and the output at the bottom. The one exception is input and output pins. These are connected to side-ways and indicate whether they are input or output by whether the wire attaches to the pointed end (input) or the flat end (output).

The main body of the code is in the rectangular box named "`HWMainLoop`". The box seen here has actual code in it so it is called a "code-box" or "leaf-box." The other kind of box is a container. One of those is called a "container-box" or "hierarchy-box." They will be seen in later programs.

Boxes can only have straight-line code with forward branches, no jumps, and no function calls in them. Thus, the wire seen going from the bottom of the main loop box to the top (via the combiner) is how loops are implemented.

The code inside the main loop uses only a few basic keywords and has syntax very similar to C and Java. The main difference to C is the semantics of the return keyword. First, execution does not end after the return statement. It continues until either the `endFn` or closing curly brace is encountered. Second, two dependent keywords accompany return: "`to`" and "`withTag`." The `to` specifies which of the outputs the value goes to. `withTag` is followed by tag code which declares the value of the tag semantically attached to the data. The tag code is enclosed in curly braces.

Thus, the loop uses the index, kept in the transport struc, to walk down the string, and send one character at a time to the "`print`" output. From there, the character flows to the "`stdout`" pin, and the underlying operating system is called to perform whatever action most closely resembles placing a character on a "generic" standard output. This could be a console window, or a GUI, or whatever the system has connected that pin to.

In this example, the compiler, semantically, constructs a transport container with a null tag to place each character into, then places that transport container onto the wire. The StdOut pin receives this container and extracts the data. However, in implementation, it is highly likely that the compilers will optimize away the transport containers and tags.

One important feature of Code Time is hinted at by the "`m`" in the update of the loop index, "`m+=1`". That's not a typo, rather it indicates that the contents of the data-container should be modified, rather than creating a new data-container with the updated value. This distinction becomes important in the presence of side-effects. When an assignment operation is performed which involves a side-effected data-container, the operation can be specified to have one of three effects on the other associations:

1. The other associations see the modification, via modifying the contents of the data-container.
2. The other associations become undefined, via discarding the original data-container.
3. The other associations are preserved and continue to see the original data-container. This association now points to a different data-container than the others.

In all three cases, the variable directly stated in the assignment expression ends up associated to a container with the values that the right-hand side evaluated to (See Part IV Imperative Language, section 3.2).

When the loop completes, the `OK` symbol is sent to the End box, which signals the virtual machine that execution has completed.

Each hardware platform has a virtual machine and a back-end compiler written specifically for it. This enables hardware-specific code-generation and optimizations, as well as hardware-specific scheduling of work. Optimizations include the replacement of tag code with direct control flow on serial machines, the free choice of whether to make data stationary (as on a single threaded machine) or actually move it between processors, the points in the flow graph at which to break up tasks and distribute them, the amount of data to send, and how many overlapping copies of each box to schedule. Part II discusses the front end and back end compilers in more detail, and Part IV Coordination Language covers tag code.

The performance of Code Time code on a single-threaded processor is expected to be better than Java and rival that of C. This is due to the ability to eliminate most, if not all, tag code, eliminate stack manipulations and register saves and restores, and implement a more efficient memory management method. The semantics allow data to remain stationary on a single-threaded machine. They also allow most memory to be systematically recycled, possibly more efficiently than malloc and free, and garbage collection needs to be performed only on side-effected structures which potentially may contain loops ("side effected" means that two or more associations to the same container exist simultaneously, at at least one point in the code). The strong typing allows the garbage collection to happen with full knowledge of the data-type and thus pointer locations. Most data is systematically freed in an orderly way (IE no fragmentation and high re-allocation of heap memory), thus the need to garbage collect becomes a very rare occurrence.

2.2 Quick Sort

The Quick Sort program illustrates the use of tags, container-boxes, many different CodeTime assignment operators, and side-effects.

The program actually looks quite busy. That reflects the nature of the Quick Sort algorithm, which has a three-deep loop structure. The two innermost loops iterate down the array, checking that each element is either larger than the pivot (top inner-loop) or smaller than the pivot (bottom inner-loop). Each loop ends at the first failure of the check, they make a swap, then the middle-loop repeats the inner ones until the top one meets the bottom one somewhere in the array. At that point, the array is split at the point where they met, and a recursive call is made. The outermost loop seen is the recursive call.

In Code Time, recursion appears to be just like any other loop. The one distinguishing feature is that recursion always has a box below the main loop which receives a duplicate of the data sent in each recursive call. This set of duplicate data is what traditionally goes onto the stack. In the Code Time data-model, the physical tracking and ordering of this data is undefined. Instead, semantically, the data collects in a pool with random ordering, arriving at random times. Time, actually, is undefined except inside a function.

In order for the algorithm to work correctly, the data has to be paired with its algorithm-required partner. This is accomplished by writing tag-code which declares the conditions the algorithm needs enforced.

The box at the bottom shows this tag code.

2.3 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 3). 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

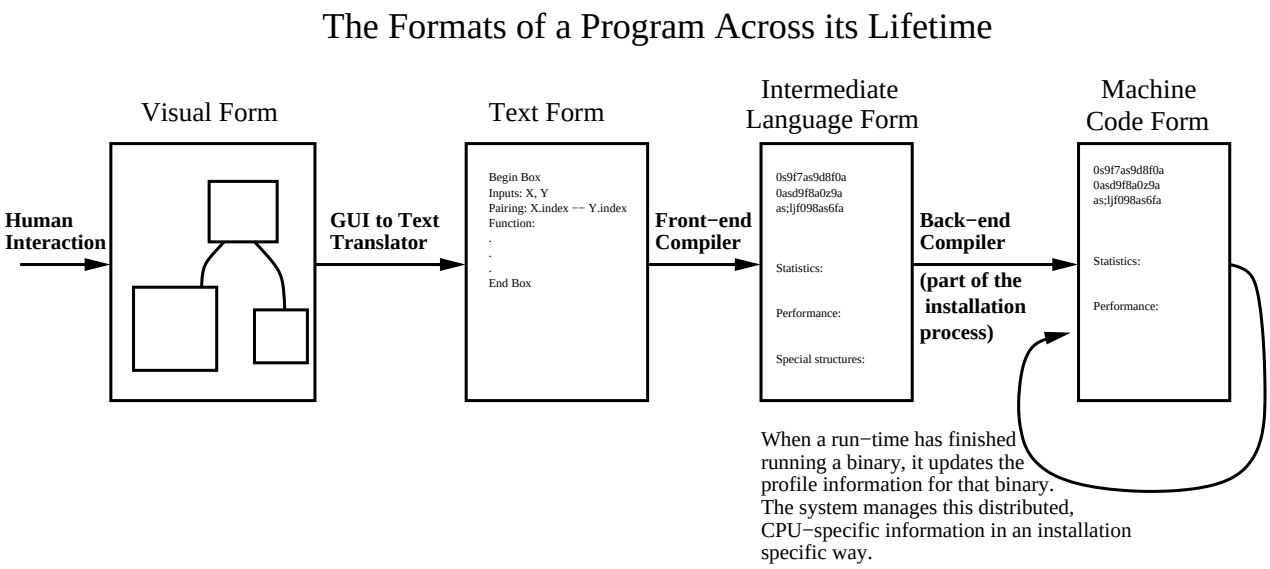

Figure 3.

2.3.1 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 3).

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

2.3.2 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.

2.4 The Code Time Data Model

In Code Time, all data resides in a "data container" (fig DataContainer). This is the basic unit of storage. No "main memory" or disk-drive or other mechanism exists. A data-container is simply a "thing" that holds data and can be passed around. Given this, variables act only as names which get associated to data containers. The name can be re-associated to different data containers. The name itself also has a physical embodiment inside a data container.

Data containers have a lifetime. Unlike traditional main memory, data containers come into existence, live for a period, then go out of existence. The virtual machine creates them, moves them around, and destroys (recycles) them at the end of their life-lines. They are created by the keyword `new` and manipulated by the various forms of the assignment operator. They can be copied, modified, have their contents transfered, and made undefined (be discarded) (see 3.2.4 for discussion of assignment operators).

All data containers have a particular shape. The shape is given via a structure declaration, which states the elements within the container. For example,

```
struct myTransportStruc
 { aVar:       int;
   aLoopIndex: int;
   aGraphNode: graphNode;
   aString:   "hello world!";
   anArray:    float[];
 }
```

declares that when a new `myTransportStruc` is created, a container will, conceptually, come into existence which has within it five names (fig myTransStruc). In addition, into existence will come two containers which hold an `int` each, an empty container which holds a `graphNode` struc, a container which holds the string "hello world!", and an empty container which holds an as-yet-unspecified-size array of `float`s.

2.4.1 The Value of the Data-Container Abstraction

Of special importance in the above is the qualifier "conceptually." In other words, the physical implementation of these semantics is unknown. In fact, physically, the back-end compiler would likely generate code that places the ints and the string directly with the pointers into a single struct. The graphNode and float array would probably be null-initialized pointers.

This separation of semantics and implementation of data structures does the following:

- supports the side-effect semantics
  - → For example, when looking at the semantics of atomic operations, having autonomous data-containers simplifies the semantic model.

- simplifies many aspects of the language
- enables tag optimizations, such as ones which eliminate physical tags
  - → In many cases, the back-end can generate code that "moves" data in a coordinated way without there ever existing physical tags, or indeed without data ever actually moving.
- discourages the programmer from C-style memory packing optimizations which would be wasted given the degree of machine-independence.
- allows the back end to manage memory efficiently
  - → For example, due to the assignment operators, the back-end knows the lifetime of every data container. Thus, it may decide to allocate one structure, then use a given location within that structure for one variable at one point in the code, and a different variable at a different point. This saves not just quantity of memory but also allows queue-based allocation and free, with calls inserted by the compiler into the machine-binary. This potentially improves performance over C-style malloc and free by eliminating fragmentation and thus virtual-memory operations. It also eliminates nearly all garbage, thereby reducing garbage collection time (garbage is only produced when containers are side-effected and the compiler cannot derive a-priori code-points where the containers become inactive).
- enhances security
  - → Because the language is distributed in intermediate format, and these abstract containers enclose all data, rules regarding usage of data-types can be verified from the "compiled" code. Thereby are prevented malicious or accidental errors due to pointer manipulation. Eventually, the application code may be run entirely in kernel mode for added efficiency. Further, the virtual-memory hardware can be safely simplified, if not eliminated.

2.4.2 Physical Meaning and Manipulation of Associations

In practice, a data container can be associated to a name or to a wire. An association has a physical embodiment. It occupies a name's data-location. For example, in the structure defined above, when the structure is created, via execution of the "new" operator, each name in the structure will be filled in with data. The data in the name-location is the embodiment of an association. The association embodiment links to the corresponding data container. In the machine-binary, this will most likely be a pointer. The compiler may opt to permanently follow the association (IE, permanently de-reference the pointer) and place a primitive data-type, like an int, in place of the name.

Associations can never be manipulated in any fashion. They can only be created or transferred. They are created via: the "p=d" or "u=d" assignment operator, the new operator, or as the result of an expression evaluation (each expression evaluation implies the creation of one or more containers which hold intermediate results). Associations are transferred via the "p=t" or "u=t" assignment operator. The virtual machine handles destruction of associations and inactive containers (see 3.2.4 for an explanation of the kinds of assignment operation).

Thus, an association is a primitive type, like a float or an int, in so much as it can actually occupy data-space in a normal data-container. However, unlike the other primitive types, associations have instead a "meta-data" meaning and cannot be modified.

2.4.3 Transport Containers and the Code-Data - Work-Data Split

Finally, an underlying concept in the language is that of "transport structures" and transport containers. Any data-container which is placed on a wire (via handing the wire an association which links to the data-container), has a tag inside of it. A transport container, then, is any data-container which has been placed directly on a wire, and therefore has a tag in it. Many structures are defined, within a program, explicitly for carrying data on wires, with no other use. These structures are called transport structures and instances of them are the most common type of transport-container. In other words, all transport-structure-instances are transport containers, however instances of other structures, which happen to get placed on a wire at some point, may also be transport-containers (these often have null tags).

Data is divided into two types: code-housekeeping data, and work data. Code-data includes things like loop indices, flags, and algorithm-specific decision-making data. Work-data includes tree nodes, matrix cell values, and arrays. In most imperative languages, like C, Fortran and Java, code-data is declared inside of functions and has a lifetime related to control-flow behavior. For example, a loop index variable has a lifetime equal to the execution of one set of iterations. In contrast, Code Time places all variables whose meaning persists past a single box instance into the data-structures which flow between boxes.

Thus, transport structures are the main carriers of code-data. Transport structures may contain associations to work-structures, but work-structures will (almost) never contain code-data. For example, it makes no sense to put a loop index into every tree node! There's no telling which boxes will perform work on those nodes, so box-specific data makes no sense. Here's an example of the definition of a transport structure and a work structure:

```
(transport struc)
structure HelloWorldStruc
 { whichChar:           int;            //loop index to walk through the string
   outputChar:          char;           //holds the value passed to the output
   helloWorldString: "Hello World!\n";  //the string that gets output
 }



(work struc)
structure TreeNode
 { parentNode:          TreeNode;
   leftChildNode:       TreeNode;
   rightChildNode:      TreeNode;
   nodeValue:           int;
 }
```

2.4.4 The Kinds of Container

Several specific types of data-container exist:

- Transport-container
  - → has a tag, code-variables, plus possible associations to work-containers
- Work-container
  - → has work-related data, such as arrays of data, matrix cell values, graph of tree nodes
- Symbol-container
  - → contains a symbol, can only be used in equality comparisons (including switch statements)
- System-container
  - → holds data generated by a system box which is meant for use by another system box
- Custom-container
  - → reserved for future additions such as possible programmer-defined container types

2.5 The Code Time Computational Model

The virtual machine has a fairly simple, basic, computation model. Data-containers flow on the wires, pool at circuit-inputs, and trigger executions of the circuits. Each execution spontaneously generates a copy of the circuit. This copy processes the particular set of inputs which triggered the execution.

An execution is triggered when a set of data-containers from the pools at the various inputs can be found which pass a filter. The filter specifies a logical relationship among the tags on the data-containers (transport-containers). Any set of inputs which satisfies this filter goes on to cause the generation of a copy of the circuit which then processes that input-set.

This pairing of an input-set with a copy of the circuit that implements a box's function is called an execution-instance. From a code point of view, an execution instance is an input-set plus a function. From a virtual machine point of view an execution instance is an input-set plus a copy of a circuit. From a code point of view, the function is applied to the input-set and produces an output on one of the box's output wires. From a virtual machine view, a data-set is chosen, generates a circuit copy, the circuit processes the input-set, and produces one or more result-data-containers. All computation takes place via this filter–>input-set–>circuit-copy–>output sequence. The two exceptions are system-box computation, which takes place in an un-defined way, and processing of exceptions.

Overall application execution begins with the `start` system-box. It emits a `<startExecution>` symbol, inside a symbol container. The arrival of this symbol triggers execution of a function, which in turn generates data-containers, which in turn trigger other function executions, and so on. Overall completion of application execution happens when the `end` system-box receives a symbol container (see the HelloWorld sample program for an example, section 2.1).

2.5.1 Input-Sets and Simultaneous Execution

In the virtual machine, data accumulates on the wires at the inputs to the function-circuits (fig ⟨reference| figInputSets⟩). The box which defined a function also declared how to choose a valid input-set for it. The process of choosing an input set involves filtering the transport containers pooled at the inputs, by using the tags inside those transport containers. The filter first declares the conditions on the tag value of each of a number of candidate inputs. Then the filter declares a relationship among the candidates which must hold. For example, simple pairing code may state that a valid input set is one piece of data (without restriction) from each input, each with the same tag value as the others (see 3.3.1 for more on input pairing code).

Every valid input set identified generates an execute-instance of the box (function-circuit). The virtual machine decides when and how these execute-instances get created and begin execution on a processor. For example, on a single processor computer, the implementation of the virtual machine may create execute instances one at a time and assign them to the CPU as they are created. Further, the back-end compiler may even have conglomerated several boxes which simply generates a single compound execution instance that combines the functions of them all. On multi-threaded hardware, however, several execute instances may be simultaneously created and run, each with different data but the same function (from the same box-instance).

Thus, conceptually, the programmer must assume that, at every box, data accumulates at the inputs and causes randomly-overlapped executions of the function.

This concept will become especially important when considering side effects. Recognising the need to control side effects depends upon understanding that each and every box could have multiple randomly-overlapped functions simultaneously executing at each and every point in time. Several side-effect-control mechanisms exist in the language, all of which involve the tags (see 3.5 for more on side effects).

2.5.2 Time in the Virtual Machine

Code Time programs have several, independent, notions of time:

- the programmer-defined "code time" embodied in the tags
- the sequential steps in the execution of the imperative language in the boxs' functions
- execution-instance ordering, via side-effect code and their interaction with the scheduler
- awareness of real-time, via the "real-time" system-box and communication with the external world through Pins

2.5.3 Special Forms

A few special forms in the computation model exist:

- system boxes

- Divider-Body-Undivider pattern
- side-effects
- combiners & splitters

System boxes are implemented directly in the virtual machine. They cannot be included, directly, in side-effects, and, in general, no assumptions may be made about ordering or overlap of their processing.

Data-parallelism is accomplished via the use of the "data-parallel" system box plus programmer specified "divider", "body", and "undivider" code. The data-parallel system box is really a conduit between the virtual machine's scheduler and application code. The scheduler is responsible for distributing data and computation across physical machines. In the act of doing this, it may decide that a particular, large, data structure needs to be divided into several pieces. This is meta-level processing (making decisions about the processing).

Once the scheduler has made a tentative initial decision on the number and preferred sizes of pieces, it performs a negotiation with the Divider (the Divider is user-supplied application-specific code). It asks the Divider what would result from a request for the preferred number of pieces. If the Divider's response is satisfactory, the scheduler then asks the Divider to perform that division. Thus, the divider is, also, meta-level processing that takes place outside the semantic flow of the program.

As for side-effect constraints, they also happen outside the semantic flow of the program, affecting instead the behavior of the scheduler. By affecting the order in which execution-instances are processed, they indirectly equate to control flow, but are not control code explicitly. Side effect constraints include specifying atomic operations, transactions, time-ordering of groups of execute-instances, box-execution triggers (for example "stop all execute instances with this tag pattern as soon as this execute instance runs"), and (syntactic sugar for tag-code) input-ordering. The default input ordering is "any", while one-at-a-time and in-order may also be specified, each of which will simply be translated to tag-code in the compiler front-end.

Finally, combiners and splitters appear in the wiring. These affect data-flow, and so take place outside of execution-instances. The splitter, for example, copies the data on its input wire to two output wires. The combiner places the data from both input wires onto a single output wire. Thus, they can be thought of as having continuous execution. An implementation may or may not perform any actual computation to effect these functions. Functionally, they may be thought of as syntactic sugar for normal boxes. However, their behavior must be considered as strictly wiring (IE, no data-pooling and no time-behavior) when considering side-effects.

See 3.6 for a fuller discussion of special forms in Code Time.

3 Syntax and Semantics of the Sub-Languages

Two alternate views of the The Code Time language are useful, one a physical view of the code organisation, the other a classification of language types. Physically, a Code Time program is made up of boxes and wires. A Code Time program is also made up of code from several different sub-languages. Each sub-language implements different classes of behavior in the program:

- An Architectural Description Language declares the actual physical structure of the program.
- An Imperative Language inside each box gives the steps of the procedure the box performs.
- A Coordination Language specifies the proper flow of data and ordering of execution-instances.
- A Specification Language attaches to each side-effect, conditions that keep the execution of that side effect safe.
- Functional aspects of the Coordination and Imperative languages enable large amounts of parallelism. This is further augmented by the Specification language enabling the parallelism inherent in side-effects.
- Special forms in the language enable many useful things, such as system functions and data-parallelism.

Each of the following sub-sections discusses in detail the semantics of one language sub-type. The first sub-section also covers most of the physical view of a Code Time program.

3.1 Architectural Description Language

The Architectural Description Language is used to state the pieces of the program and the connections between the pieces.

The pieces are (fig ADLpieces):

- Box-type definitions
  - → only types of code-box can be defined; code boxes contain the bulk of app code
- Data-container-type definitions
  - → both transport structure definitions and work-data structure definitions
- Instances of
  - → code boxes, each of a type defined in the Box-type definitions
  - → container boxes, into which are placed inside code box instances
  - → system boxes
  - → data-container-type definition boxes, which are placed inside container boxes
  - → combiner and splitter instances, which are also placed inside container boxes
- Wires
  - → These connect the output of one box-instance to the input of itself or the input of another box-instance. The box instance the wire comes from may be either a code box instance or container box instance, likewise for the box instance the wire goes to. A wire may also connect the input of a container-box to the input of a box inside itself, or the output of a box inside itself to a container box's output. The internal boxes may be either code boxes or other container boxes.

3.1.1 Text Format of the ADL

In the GUI format, the Architectural Description Language takes the form of graphical elements. However, in this sub-section, the text form of the ADL is discussed. This is the form which is fed into the front-end compiler. This text form has four kinds of declaration:

- box type declaration
- box instance declaration
- combiner or splitter declaration
- wire declaration

3.1.2 Code-Box-Type Definition

A box instance declaration specifies the type of box (fig ADLBoxDecl). Pre-defined types exist, for example system box instances are all of pre-defined types. Only code-box instances may be of a type defined by the application programmer. Each available type of code-box must be defined. This type definition specifies all of the internals of the code-box, including:

- name of the type being defined
- name and data-type of each input to an instance of that box-type
- name and data-type of each output from an instance of that box-type
- ordering constraint on each input (optional) – this is syntactic sugar for tag code
- execution-instance ordering constraints (optional) – this defines groups of execution-instances and declares time-ordering among the groups.
- trigger actions (optional) – this defines groups of execution-instances of the box, associates each group to a trigger-name and to a meta-action to be performed when that trigger takes place

- trigger definitions (optional) – trigger criteria, involving transport container tags, may be defined which involve
  - → an arrival to an input
  - → the execution of a particular execution-instance
  - → the generation of an output

  satisfying the criteria causes the named trigger to take place
- pairing code – defines a filter which declares what constitutes a valid input-set
- function – the imperative code which defines the data-transformation performed by the box
- side-effect code – inside the function, any assignments involving side-effected data-containers may have constraints added which state the conditions under which it is safe to process an execution-instance of the box.
- tag generation code – inside the function, returns made to outputs may declare operations to be performed on the tag of the transport structure being output.

Here is a sample code-box type-declaration:

```
PrepareForSearchBoxType hasForm
 { inputs   { array: int[], size: int }
   outputs { done: BinSearchStruc }
   inputOrdering{ any, any }
   inputPairing { any }
   boxInstanceOrdering { any }
   function:
   PrepareForSearch( array, size )
    { //the contents of the function go here.. see Imperative Language,
      // Coordination Language, and Specification Language
      sideEffectableVar.field m=t tempData asPartOfAtomicOp myAtomicOpName;
      return searchResult to output withTag
       { //tag generation code
       }
    }
 }
```

3.1.3 GUI vs Text Format Differences

One important difference between the text-format and the graphical format is that in the graphical format, a box has no type declaration. Any place a box instance appears is also the definition of the box type. In other words, any place a box is viewed, both the instance declaration and the type definition are seen. The type itself may be edited from any viewed instance. That modification of the type propagates to all instances of that type. When a code box is copied, the programmer specifies whether they wish to share the type with the original, or to make a copy of the type as well. If a copy of the type is made, the code in the new box-instance can be modified without affecting the source of the copy. GUI commands are defined as part of the developer environment specification which show all boxes sharing the type of a selected box.

Inheritance of box-types is implemented in the GUI environment via copying a box instance as well as its type. , when say the name, say the base type then say the new type name.. with dot notation like Java..

then, get the original boxes in one color, can modify code or re-define entire box, in second color, each generation the color changes..

can pull up a window to see box types and sub-types already defined (for start, just use tree-view)

in text version, spec the name in same dot notation, then give structural declarations (say “replace box” plus old box name and new box name (including inheritances))

Because every box is the same as every other box in the GUI, means there are no code-dependencies created by the inheritance structure when grab a box to use elsewhere (however, MUST use the GUI!!! The text version does not have this property)

only a GUI can have this "distributed" type-definition property. This means also that only in the GUI can code be grabbed for re-use without worrying about file-dependencies.

3.1.4 Container-Box-Type Definition

Once code-box types have been defined, container-box types are defined (fig containerTypeDef). The definition of a container-box-type declares, within it, instances of other box-types, including other container-box-types which may have not been defined yet.

The declaration of a box instance inside a container box includes the name of the instance, the name of the type, and a wiring for each input and each output (even though this causes each wire to be declared twice).

Container boxes are also where all data-structure-box instances reside. This allows the definition of any structure used in any box to be located by traversing the ancestors of that box. A side-effect of this organisation is that the definition of each data-structure (including transport structures) used in a particular box must be declared in some ancestor of that box, causing duplicate data-structure-box instances and correct placement of these instances. The GUI tool manages this duplication and placement automatically when coding in the graphical format.

Here is an example of a container box:

```
BinSearchBox contains
 { PrepareForSearch1: PrepareForSearchBox withWiring
    { array:In wiredTo BinSearchBox::array;
      size:In wiredTo ArraySizeBox1::size;
      done:Out wiredTo CheckMiddleElem1::S;
    }

   ArraySizeBox1: ArraySizeBox withWiring
    { inputArray wiredTo BinSearchBox::array;
      size wiredTo PrepareForSearch1::size;
    }

   struct myTransportStruc
    { aVar: int;
      aLoopIndex: int;
      aGraphNode: graphNode;
      aString: "hello world!";
      anArray: float[];
    }

   <snip>
 }
```

BinSearchBox is the name of the box-type, the "`contains`" keyword indicates that it's a container-box type-declaration, and the instances declared inside of it indicate the contents. Note that instantiation is hierarchical. This means that none of the instances inside BinSearchBox will actually appear in the program yet. It will only be when an actual instance of BinSearchBox is created, that the instances inside this box will be created. The top level of the program is the top of the instance-creation hierarchy.

3.1.5 Declaring the Top-Level of a Program Hierarchy

Here is an example of the top-level declaration of a program:

```
Program BinSearch contains
```

```
{ StartSystemBox1: StartSystemBox withWiring
   { out:Out wiredTo GetArrayToSearch1::goForIt;
   }

  GetArrayToSearch1: GetArrayToSearch withWiring
   { goForIt:In wiredTo StartSystemBox1::out;
     array:Out wiredTo BinSearchBox1::array;
   }

  BinSearchBox1: BinSearchBox withWiring
   { array:In wiredTo GetArrayToSearch1::array;
     position:Out wiredTo ShowPosition1::position;
   }

  <snip>
}
```

This is exactly the same as a container-box type declaration, except for the "`Program`" keyword which specifies that this is the top-level. Note the "`StartSystemBox`". This system box must be included in every Program declaration. It emits a dummy-value as the first activity which begins execution of the program. Likewise, the last activity in the program should output to the "`EndSystemBox::in`" input. This tells the system that the program has completed in a clean way. Whatever is sent to the EndSystemBox will act as the program completion-status indicator. Special symbols are defined as status-indicators which can be sent. Only one of these special symbols may be sent to the EndSystemBox (they may be translated to machine-OS-specific values by the virtual machine or back-end compiler).

All the instances declared inside the Program container are actually created, and all instances declared inside each of those are in turn created, and so on recursively.

3.1.6 Scope of Instance Names and the Effects of Side-Effects

The same instance name can be used inside different container declarations. No instance names declared inside a container can be seen outside the container-box nor at lower levels inside the container-box, except by side-effect code. For side-effects, names are resolved by prepending the container-box instance name with dots. For example "`HigherContainerBox1.MyCodeBox4::inputA.tag.value`" specifies that the quantity of interest is the value of the tag of the transport structure bound to the inputA position of the input-set of an execution instance of a neighboring box, called `MyCodeBox4`, sharing the same container as the box that this specification appears in.

An important aspect (side-effect?) of this scheme is that side-effect code cannot be generic. This is appropriate to the nature of side effects. When code containing side-effects is reused, the smallest amount of code that can be taken is the lowest-level box enclosing all instances named in any side effects in the desired code, plus any side-effects in any of those instances, and so on.

Note that this side-effect cross-box dependence is the only constraint on breaking up code. Other than side-effects, any division of code may be made for reuse purposes. The GUI tools enforce the side-effect code-partitioning constraint. The tools include many features to highlight, discover, modify, and manage side-effects which cross box boundaries.

3.1.7 System Boxes

System boxes come with their type already defined. Only instances of system boxes may be declared. This is done in the same way as declaring an instance of any other kind of box. Typical system box types include:

- StartSysBox
  - → emits a dummy value to initiate program execution
- EndSysBox
  - → a symbol sent to End tells the system that program execution is complete

- LookUpResourceSysBox
  - → translates a generic string name into a physical location and mechanism, which are returned inside a system-container. System-containers cannot be looked into nor modified.
  - → the string name should be generic like "CurrentUserPreferences" which the virtual machine or back-end compiler translates into a file name plus a machine name or URL or other OS and HW specific info
  - → the URL or other HW specific info is given to application-code inside of a system-container, which allows the info to be passed around, given to system boxes which perform generic queries, etc, without exposing any machine details to the application code.
- InitializeStreamSysBox
  - → takes in the result of a resource lookup and begins emitting data from the resource
  - → for example, this may be back-end compiled as a file open plus a thread which continuously reads from the file, in synchrony with the the virtual-machine's run-time which gets told when data is waiting, consumes a chunk, gets told when more is waiting, etc – the hand-shaking is implemented by the back-end compiler and happens behind the scenes
- CloseStreamSysBox
  - → closes a previously initialized stream, which causes data to stop flowing out of it. Side-effect code may be used to cause this to happen at a specific data value, or some other "time" related point.

Note that at the time of this writing System boxes are still under development. More will be added, current ones will be modified and clarified.

3.1.8 Combiners and Splitters

Last of the elements are combiners and splitters. These are included in the wiring inside container-boxes. Three kinds of splitter exist, one kind creates duplicate associations (DuplicatingSplitter), one creates a copy of the top-level transport structure and first-level data-containers (TopCopyingSplitter), and the third creates a recursive copy of every data container which can be reached from the top-level transport structure (RecursiveCopyingSplitter).

Graphically:

- the chevrons and upside-down chevrons in the wiring are combiners and splitters
- A combiner has two inputs at the top and one output at the bottom. It acts as a "zero-delay" architectural element which simply takes any data that appears at either input and presents that data at the output. Relative ordering of one side vs the other is undefined.
- A splitter has one input at the top, and two outputs at the bottom.. It acts as a "zero-delay" duplicator. An annotation specifies the kind of duplication:
  - → "c" indicates copy of the top-level structure and first-level data-containers (with contents)
  - → "rc" indicates recursive copy of all data containers reachable from the top-level structure
  - → "d" indicates duplication of the association, which causes all associations to the transport strucs which pass through the splitter to be turned into side-effectable associations. Side-effectable associations require safety specifications. These specifications must be added to writes that modify the contents of a data-container reached via a side-effectable association.

More complex kinds of copies may be implemented by writing a custom code-box to perform the copy. In fact, combiners and splitters are just syntactic sugar for pre-defined (library) code-box-types.

Here is an example of the ADL declaration of combiners and splitters:

```
MyContainerBoxType contains
```

```
{ ABoxInstance1: ABox withWiring
   { anInput:In wiredTo combiner1::out;
     otherInput:In wiredTo splitter1::outA;
     anOutput:Out wiredTo splitter1::in;
   }

  combiner1: Combiner withWiring
   { inA:In wiredTo foo1::barOut;
     inB:In wiredTo yada2::barOut;
     out:Out wiredTo ABoxInstance1::anInput;
   }

  splitter1: DuplicatingSplitter withWiring
   { in:In wiredTo ABoxInstance1::anOutput;
     outA:Out wiredTo ABoxInstance1::otherInput;
     outB:Out wiredTo fooBar1::inputA;
   }

  splitter2: TopCopyingSplitter withWiring
   { in:In wiredTo SomeOtherInstance1::itsOut;

  splitter3: RecursiveCopyingSplitter withWiring
   { in:In wiredTo StillOtherInstance1::stillOtherOut;

  <snip>
}
```

3.1.9 The Connectors and Relative Ordering of Data

Connections are made between boxes by wires. A wire, semantically, indicates the flow of data. It is connected between a box output and a box input (except inside container-boxes where a wire may connect an internal box input to one of the container's inputs, or similarly for outputs). Each time an execution-instance produces an output via a "`return to output`" statement, the value travels on the wire attached to that output and arrives in the input-pool at the other end, where the wire attaches to an input. The input-pairing code gathers input-sets from these input-pools of data, and the scheduler subsequently turns these input sets into execution-instances and schedules these for processing.

In general, no ordering is defined on any wire nor between wires. In fact, two "sequential" outputs may not arrive at the other end of the wire in the same order as they were generated. Each execution-instance defines its own time-line. Time (and therefore ordering) is not defined between separate execution-instances. So, no such thing as "sequential" outputs from a single box exists when the outputs are generated by different execution-instances.

3.2 Imperative Language

The Imperative portion of the Code Time language comprises the core of the language. The function in each code-box is written in the imperative portion of the language and augmented by the specification language (for side-effects) and the coordination language (tag generation).

The Imperative language is imperative in the sense that its semantics have the notion of time ordering of steps, as well as the notion of memory-locations. Each line of the imperative code is ordered relative to the other lines. The operations specified by a line take place after the preceeding line and before the following line. All data specified in operations has the semantics of physical embodiment in a physical container (see the section on the data model in part III).

The Imperative language appears only inside a code-box and a subset of it inside a structure-specification box. It is used to state the sequence of steps that carry out the computation of the box's function. It has relatively few keywords and operators, based upon the syntax of C and Java.

3.2.1 Imperative Language Keywords

The list of keywords:

- `If-then-else`
- `switch-case`
- `return-to-withTag`
- `endFn` (or possibly one of these alternatives: `break`, `stop`, `stopEx`, `endEx`, `end`)
- `new`

The groups of operators:

- arithmetic, shift and bit-wise logic
  → `+, -, *, /, mod, &, ^, <<, >>`
- comparison and logical-combination
  → `==, <, <=, >, >=, &&, ||`
- assignment
  → `p=c, m=c, u=c, p=rc, m=rc, u=rc, p=t, m=t, u=t, p=d, u=d, p=s, m=s, u=s, m+=, p+=` (`+` or any two-input operator)
- accessor
  → `[], .`
- grouping
  → `(), {}`
- delimiter
  → `;`
- special forms
  → `//, /*, */, [--, --], "", '`

Each of these should be familiar, with the exception of:

- `return-to-withTag, endFn, new`
- the assignment operators
- some new uses of `{}`
- `[--` and `--]` for comment start and end

3.2.2 The Differences of Code Time Semantics

The semantics of data-behavior of all these keywords are different than other languages. All data in Code Time exists in a data containers, including intermediate results created during the evaluation of an arithmetic expression. Thus an arithmetic expression such as "A + B" results in an association to a data container holding the sum. The container holding the final result of the expression may be associated to by a name (variable). The fate of this expression-results-container depends upon the form of assignment operator used, as elaborated further below (3.2.4) and in the section on the data model (2.4).

3.2.3 return-to-withTag, endFn, new

The return-to-withTag triplet causes a value to be placed on an output with a tag attached. The main difference between Code Time's return semantics and that of other languages is that it does not imply the end of execution of the function. Rather, many values can be output from a single function invocation, so execution continues until either the "endFn" or final curly brace of the function is encountered.

As an example, "`‘return foo to out1;`" sends whatever's associated to foo on the wire attached to `out1`. Because the `withTag` keyword has been left out, `foo` is required to be associated to a transport-container which was input to the box. The tag of that transport-container is not modified. Note that execution does not cease after executing this line, rather it continues until either an `endFn` or the enclosing brace of the function is encountered.

The "`new`" keyword creates a data-container dynamically. It is followed by a container type and results in an association to a new container of that type.

For example "`new myTransportStruc;`" creates a data container for the struc itself plus empty containers for any associations to containers inside the struc. See the section on the data model.

3.2.4 The Forms of Assignment

Some aspects of the forms of assignment are also discussed in the section on the data-model. The reason for the multiple kinds of assignment is to allow the programmer control over the disposition of the data-containers involved in the assignment operation. Every assignment involves one, two or three data containers. The left-hand side of an assignment always specifies a variable, which usually already has a data-container associated to it. The right-hand side always evaluates to an association to a data-container. When the assignment operator specifies a copy, the copy made is the third container involved in the assignment.

An assignment states the post-assignment disposition of each data-container involved in the assignment operation.

For the left hand side container, this can be one of:

- "`m`" – modify the contents of the pre-assignment existing data-container
- "`p`" – preserve the pre-assignment data-container as-is and associate the left-hand-side variable to whatever results from the right-hand-side of the assignment operation.
- "`u`" – undefine the left-hand-side container after the assignment is complete, making it cease to exist. This is equivalent to discarding the pre-existing left-hand-side data container.

For the right hand side, the behavior is one of:

- "`t`" – transfer the container from the right-hand-side to the left hand side. Exactly what transfer means depends upon the behavior specified for the left hand side. However, in all cases, when the assignment operation is complete, the right hand side no longer has a valid association. For example, if the right hand side is a variable, then that variable no longer contains a valid association when complete. If the right hand side is an operation, then transfer is *required*. If the right hand side is a symbol then "t" cannot be used. Here are the cases:
  - → "`m=t`" means transfer the contents from the right hand side to the left hand side container, then the right-hand-side container ceases to exist. However, if the right-hand-side container was side-effectable, then the execution of this assignment ends this branch of the container's life-line. When all of a container's life-lines have been ended, the container ceases to exist.
  - → "`p=t`" means transfer the association from the right-hand-side to the left hand side. The container that the left hand side used to associate to is preserved. This does not end the container's lifeline, as the container has not been affected, only the association to it was moved.
  - → "`u=t`" means transfer the association. The previous left-hand-side container ceases to exist. Again, the right-hand-side-container's lifeline continues.
- "c" – copy the top-level right hand container. Again, the exact behavior depends upon what was specified for the left hand side. In all cases, the right-hand-side container continues to exist, and the lifeline of the right-hand-side container is unaffected. The "c" form can only be used with a variable on the right-hand-side:
  - → "m=c" means copy the contents of the right-hand container to the left hand container.

  - → "p=c" means make a new container with identical contents to the right-hand-side container. Then, associate the left-hand-side variable to this new container. The container the left-hand-side previously associated to is preserved.
  - → "u=c" identical to "p=c" except that the container the left-hand-side previously associated to becomes undefined (is discarded).
- "rc" – recursive copy. This is identical to "c" except that all containers reachable from the right-hand-side association are copied, rather than just the first container immediately associated to.
- "d" – duplicate the association. This causes a duplicate association to the right-hand-side data-container to be made. This is the mechanism by which side-effects are enabled. The exact behavior depends upon the left-hand-side:
  - → "m=d" is not allowed. This makes no sense semantically.
  - → "p=d" associates the left-hand-side variable to the data-container also associated to by the right-hand-side variable. The right-hand-side must be a variable. The container the left-hand-side used to associate to is preserved.
  - → "u=d" is the same as "p=d" except that the container the left-hand-side used to associate to becomes undefined (is discarded).
- "s" – specifies that the right-hand-side is a symbol. This kind of assignment creates a symbol-container.
  - → "m=s" changes the contents of the existing left-hand-side symbol-container
  - → "p=s" preserves the container the left-hand-side used to associate to
  - → "u=s" undefines (discards) the container the left-hand-side used to associate to

The special case of an assignment that states or implies the same variable on both the left and right-hand sides only states the disposition of the container associated to the left-hand-side variable. That data-container is either preserved and a new association is made to the container holding the result of the right-hand-side operation, or the data-container associated to by the left-hand-side variable is modified to contain the result of the right-hand-side operation. For example, "`myVar m+= 1;`" causes the data-container associated to myVar to have its contents incremented by 1.

Assigning a symbol to a variable causes the variable to be associated to a special-form of data-container. This kind of data-container cannot be used in arithmetic or other operations. It also cannot be created or destroyed. The only kinds of operations allowed are assigning a variable to associate to such a container and to test identity of the container. Thus, only the disposition of the left-hand-side container is given in an assignment to a symbol.

Note that the front-end compiler is able track the life-lines of all data containers, which enables the back-end to implement reuse their storage. Even if the "t" indicator is not used, the front-end compiler can still often deduce the life-lines of data containers and enable reuse of their storage.

3.2.5 New Uses of the Curly Braces

The `{}` have some new uses in Code Time. In general, they still delimit a block of code. However, for side-effect specifications they also indicate grouping.

For example,

```
1 S.array[ S.upperEmptySlot ] m=c S.pivot withOrdering
2  { thisLevel s= this union othersWith{ other.tag.level == this.tag.level };
3    higherLevel s= othersWith{ other.tag.level > this.tag.level };
4    doAllOf{ thisLevel } before doAnyOf{ higherLevel };
5  }
```

The first and last curly braces delimit a block of code, whereas the other four pairs delimit groups of things.

3.2.6 New form of Comment Delimiter

Finally, a new form of comment start and end has been introduced simply for typing ease and visual cue clarity. The "[", "]" and "-" symbols are all non-shifted on most keyboards, with the "-" just above the "[" and "]" making it easy and quick to type the comment start and end. Also, the sequences "[--" and "--]" have a strong visual cue indicating which side of the delimiter the comment lies on.

3.3 Coordination Language

The Code Time Coordination Language implements the data-flow behavior of the inter-box communication. It coordinates the ordering of execution instances and the grouping of data such that the algorithm's constraints are observed. It has three parts:

- tag generation code
- tags attached to data (semantically)
- input pairing code (declares what constitutes a valid input set)

Tags are generated as part of a return-to-withTag compound statement. Every top-level container leaving a box has a tag attached to it, either via the withTag portion of the return statement, or by the default behavior which simply leaves the tag unchanged (of a transport container passed through from the input). The front-end compiler will catch if a tag has not been initialized, or if a tag field is accessed that doesn't exist (say, an un-copied tag field of an ancestor to the current transport struc).

Here's an example:

```
return newRight to rightRecurse withTag
 { tag u=rc S.tag;
   tag.self.level m+= 1;
   tag.self.type u=s R;
   tag.parent u=c S.tag.self;
 }
```

This code is taken from the quicksort sample program. It returns the data-container associated to the newRight variable to the "rightRecurse" output, and attaches a tag (fig newRightTag).

It first copies the tag from the input transport-container, discarding whatever might have previously been in the tag location of the container associated to the newRight variable. The newRight variable happens to have a newly-created transport struc in it. The "u" on the left-hand side of the assignment, therefore, causes the uninitialized container to become undefined (IE, it is discarded). The "rc" on the right-hand side causes a full recursive copy of the tag of the "S" transport container (from the input) to be created and an association to the tag-copy placed into the tag of the newRight container. Next, the value of the copy's self.level field is incremented (via modification due to the "m" on the left-hand-side). Then the self.type field is associated to a symbolic container holding the symbol "R" (undefining, and thus discarding, whatever self.type previously associated to). Finally, the parent field of newRight's tag is associated to a copy of the container associated to by the input transport container's tag's "self" field .

The tags themselves have their fields defined by the programmer within the tag-generation code. Only a few restrictions exist:

- Any data flowing into an "in-order" input must have tags that start at 0 and increase by 1, with no two transport strucs reaching that input having the same tag value, and no tag values being skipped.
- When pairing code is specified, every transport struc which comes into one of the inputs specified in the pairing code must have a tag with a unique pattern of field-values.

For practical implementations it is good, from a performance perspective, to have tags such that each field's value has a simple mapping to an in-order integer sequence which begins at 0. Tag fields with a finite domain of values fit this (equivalent to enumerated types), as well as tag fields assigned from mathematical expressions that can be symbolically manipulated by a relatively un-sophisticated symbolic manipulation package (embedded in the front-end compiler) such that the results of the expression, passed through the symbolic manipulation, form an in-order sequence starting at 0.

3.3.1 Input Pairing Code

This code appears near the top of a box (graphically) and near the top of the box type-declaration (in text form). It declares a filter. Any set of transport containers from the inputs which passes the filter is a valid input-set. A valid input-set has the box's function applied to it. This application of the box's function to an input-set is the basic unit of computation in Code Time.

The filter takes the form of a logical combination, using AND and OR operators, of comparisons between mathematical expressions involving tag values.

Pairing code has the following parts:

- keyword: "pairingCode"
- curly-brace delimited block
- a selector for each element of the input-set
  - → the selector uses the keyword "oneFrom" followed by a curly-brace delimited list of inputs
  - → semantically, a single transport container is chosen from one of the inputs in the list
- an optional constraint on the tag of the transport container chosen,
  - → introduced by the "suchThat" keyword
  - → this constraint filters which transport containers can bind to that input-set position, using comparison, logical combiners, and arithmetic, on the candidate's tag
- an expression stating the constraint between selected, candidate, transport containers
  - → this expression is introduced by the "isAValidSetWhen" keyword
  - → uses comparison, logic, and arithmetic operators between tags of different containers

Here is an example from the quicksort program:

```
pairingCode
 {
   parent       := oneFrom { new };
   leftChild    := oneFrom { done } suchThat leftChild.self.type == L;
   rightChild   := oneFrom { done } suchThat rightChild.self.type == R;

   isAValidSetWhen
   parent.self == leftChild.parent && parent.self == rightChild.parent;
 } presentSetAs { parent, leftChild, rightChild };
```

This code selects one transport-container from the "new" input, semantically at random. It then picks one from the "done" input whose tag contains "self.type" with value "L", and another transport container from the same "done" input, this time with the "self.type" tag having the value "R".

Given this set of transport structures chosen, semantically, at random from those available in the input data pools, the inter-container filter is then applied. In this case, if the transport-container from the "new" input has the same value in the "self" portion of its tag as both the transport containers from the "done" input have in the "parent" portion of their tags, then the input set passes the filter. Note that this particular input set happens to have three elements which were taken from only two inputs.

3.4 Functional Aspects

Code Time as a whole has many of the characteristics of a functional language. It has the property that the order of executing functions (boxes) is independent of the value of the result obtained (subject to the constraints specified in the coordination language and specification language). This enables easy extraction of parallelism.

In particular, the coordination language implements data-flow style semantics between boxes. Coupling this data-flow style with the lack of persistent variables in the imperative language, gives rise to this execution-order independence. This combination can be seen, in one sense, as the result of applying referential transparency. In other words, if one took a functional language and replaced each function call with a box containing the function, then represented the call and return as wires which moved the parameters and results, the result would look very much like Code Time code.

The most important aspect of Code Time's ordering independence is the ability for multiple execution-instances of a box to be simultaneously computing. Any time a valid input set is available, an execution-instance, which applies the box's function to that input set, may be run. This execution is subject only to side-effect constraints. For example, a large collection of input-sets may be saved up, sent to a remote machine as a bundle, and run all in parallel on different threads of a multi-threaded processor, or different CPUs of a multiple CPU machine.

In fact, this ability to pool input sets then execute them all in parallel generalizes one form of classical data-flow semantics which wait for data to be accepted before sending more. The combination of these semantics and the enabling, in a functional setting, of the parallelism inherent in side-effects increase the availability of parallelism, increases the choice of parallelism granularities, and thus increases the efficiency possible when mapping a given program onto a given hardware configuration.

3.5 Specification Language

The Specification language in Code Time is used to enforce the safety of side-effects. Side effects provide a large amount of parallelism, which is lost in languages lacking them (consider shared data-structures on shared memory machines). Code Time's specification language attempts to enable declaring the minimum constraints possible which affect the safety of side-effects.

The mechanisms provided to specify side-effect related constraints include:

- explicit ordering of groups of execution-instances
- atomic operations
- transactions
- some types of pairing code, such as input-ordering (one-at-a-time, or in-order)
- triggers which are tripped as the result of the processing of one execution-instance which then affect the processing of other execution instances

Explicit ordering of groups of execution instances has two portions. First, filters are declared which select groups of execution instances. Second, time ordering of those groups is declared. The keywords involved are these:

- withOrdering
- this
- union
- intersection
- othersWith
- doAllOf
- doNoneOf

- doAnyOf
- before
- during
- after

Side effect constraints include specifying atomic operations, transactions, time-ordering of groups of execute-instances, box-execution triggers (for example "stop all execute instances with this tag pattern as soon as this execute instance runs"), and (syntactic sugar for tag-code) input-ordering. The default input ordering is "any", while one-at-a-time and in-order may also be specified, which will simply be translated to tag-code in the compiler front-end.

3.6 Special Language Features

Code Time has three main special forms:

- System Boxes
- Pins
- Divider and UnDivider

3.6.1 System Boxes

System boxes allow hardware-independent specification of operating-system services. Most of them are still under development at the time of this writing.

3.6.2 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.

3.6.3 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).

4 Conclusion

At this point, a programmer should have the basic knowledge needed to understand the sample programs and modify them for their own purposes.