The Concurrent Collections Programming Model

Zoran Budimlić[†] Michael Burke[†] Vincent Cavé[†] Kathleen Knobe[‡] Geoff Lowney[‡]
Jens Palsberg[⋆] David Peixotto[†] Vivek Sarkar[†] Frank Schlimbach[‡] Sağnak Taşırlar[†]

[†]Rice University
[⋆]University of California at Los Angeles
[‡]Intel Corporation

Abstract

We introduce the Concurrent Collections (CnC) programming model. In this model, programs are written in terms of high-level operations. These operations are partially ordered according to only their semantic constraints. These partial orderings correspond to data dependences and control dependences.

The role of the domain expert, whose interest and expertise is in the application domain, and the role of the tuning expert, whose interest and expertise is in performance on a specific architecture, can be viewed as separate concerns. The CnC programming model provides a high-level specification that can be used as a common language between the two experts, raising the level of their discourse. The model facilitates a significant degree of separation, which simplifies the task of the domain expert, who can focus on the application rather than scheduling concerns and mapping to the target architecture. This separation also simplifies the work of the tuning expert, who is given the maximum possible freedom to map the computation onto the target architecture and is not required to understand the details of the domain. However, the domain and tuning expert may still be the same person.

We formally describe the execution semantics of CnC and prove that this model guarantees deterministic computation. We evaluate the performance of CnC implementations on several applications and show that CnC can effectively exploit several different kinds of parallelism and offer performance and scalability that is equivalent to or better than that offered by the current low-level parallel programming models. Further, with respect to ease of programming, we discuss the tradeoffs between CnC and other parallel programming models on these applications.

1. Introduction

It is now well established that parallel computing is moving into the mainstream with a rapid increase in the adoption of multicore processors. Unlike previous generations of mainstream hardware evolution, this shift will have a major impact on existing and future software. A highly desirable solution to the multicore software productivity problem is to develop high-level programming models that are accessible to developers who are experts in different domains but lack deep experience with parallel programming. A high-level programming model can be designed so that its semantics provide a basis for proving properties that simplify the understanding, debugging, and testing of a parallel program.

In this paper we introduce the Concurrent Collections (CnC) programming model, built on past work on TStreams [20], which enables programs to be written in terms of high-level operations, which are partially ordered according to only their semantic constraints, corresponding to data and control dependences.

This approach supports an important separation of concerns. There are two roles involved in building an application. One is the role of a domain expert, the developer whose interest and expertise are in the application domain, such as finance, genomics, or numerical analysis. The other is the tuning expert, whose interest and expertise are in performance, including performance on a particular platform. These may be distinct individuals or the same individual at different stages in the application development. In some cases the tuning expert may even be software, such as a static or dynamic optimizing compiler, or a runtime. CnC is a high-level specification language that the domain and tuning expert can use to facilitate cooperation and raise the level of their communication. This high level discourse supports an iterative process where the two experts can make adjustments to their specification/code based on their conversations. Even if the task and domain programmers are the same person, it can allow the programmer to focus on domain issues at one point in time and to focus on tuning later.

Writing a specification in this model does not require an understanding of the details of the target architecture, which simplifies the task of the domain expert, who is concerned primarily with his/her area of expertise, and the semantics of the application. She precisely specifies, at a high ("white board") graphical level, the operations (steps) and the semantic ordering constraints required by the application, producing a CnC specification.

This separation also simplifies the work of the tuning expert. The tuning expert is given the maximum possible freedom to map the computation onto the target architecture and is not required to have an understanding of the domain. An important consequence of the separation of concerns is that CnC is highly portable.

The high-level, declarative specification of semantic constraints supports the formulation of a relatively simple formal semantics for the model. Key design decisions keep the semantics simple, making it possible to prove some useful properties of the model. We have chosen that programs in the model conform to a dynamic single-assignment condition. This is a basis for making program execution provably monotonic with respect to program state, which is a basis for making the model provably deterministic. Deterministic execution of parallel programs simplifies program analysis and understanding, facilitating compiler optimization, testing, debugging, and tuning for a target architecture. For these reasons languages have aspired to establish confluence of various properties leading to determinism.

We present three current implementations of CnC: one built on top of the Intel Threading Building Blocks (TBB); one built on top of the `java.util.concurrent` library (JUC); and one built on top of .NET.

The contributions of this paper are:

1. A description and rationale of the main choices in the design of the CnC programming model;
2. A formal description of the execution semantics for CnC, and a proof that this model guarantees deterministic computation;
3. Experiments to demonstrate:
   (a) The CnC model can express a wide range of parallel programming idioms;
   (b) CnC can effectively exploit several different kinds of parallelism and offer performance and scalability that is equivalent to or better than what is offered by the current low-level parallel programming models;
   (c) Tradeoffs with respect to ease of programming between CnC and other parallel programming models.

Section 2 provides a rationale for the most fundamental choices in the design of CnC. Section 3 presents the main features of the language, and provides an example to illustrate the process of creating a CnC graph for a specific application. Section 4 describes the semantics of a Featherweight CnC language, and proves that the language is deterministic. Section 5 describes the implementation of CnC on several platforms. Section 6 discusses several experiments that demonstrate the efficiency and usability of CnC. Section 7 describes related work. We conclude in Section 8.

2. Design Rationale

CnC is a model not a language. There are two relevant distinctions:

- We are not concerned about syntax. There are three ways of writing a specification: a textual representation of the CnC graph, use of a class library that implements CnC , and a GUI.
- Within the model different language implementations can make distinct choices that arise from tradeoffs concerning ease-of-use, efficiency, generality, and analyzability.

The three main design choices for CnC are:

1. To create a high-level interface between the domain and tuning experts;
2. To require determinism;
3. To have three fundamental constructs: data, control, and step collections.

The creation of a high-level interface between the domain and tuning experts supports the goal of a separation of concerns between their work. In the context of this paper, this separation of concerns means several distinct things:

- The domain programmer can focus on the specification without having to think much about scheduling or performance.
- The tuning programmer can focus on performance without having to learn much about the domain.
- The domain programmer and the tuning programmer can communicate through use of the CnC specification as the basis of the discussion, raising the level of this discourse.
- The tuning expert has maximum flexibility in mapping to the target.
- The same CnC specification can be used for mapping the application to distinct target architectures.

With the current CnC specification, we have made significant progress toward the above goals. We are for the most part assuming that the domain expert will not be tuning the application. But recall that the tuning expert might be the same person as the domain expert at a different time, a distinct person, a static analyzer, or a dynamic runtime system. Consider the following aspects of tuning: grain size, mapping steps across address spaces (distribution), mapping steps across time within an address space (scheduling), and mapping data to memory locations. Without addressing all possible combinations of tuning issues and the type of tuning expert, we now discuss a few interesting scenarios.

First we consider a static analyzer that tunes the grain size and all of the mapping issues for a CnC specification. Compare this with a static analyzer that has to analyze a serial program. For a static analyzer, the producer, consumer and control relationships, including the tag functions, constitute most of what is needed for the analysis. The specification directly provides this information. Other information that might be useful includes execution times for the steps and size of the data instances. But the actual code within the steps in not relevant.

Starting from the serial code, the static analysis would have to deal with unnecessary constraints, including:

- Overwriting of locations with distinct values;
- Arbitrary (not semantically required) orderings;
- Tight binding of two control flow questions: *if* the program will execute some code and *when* it will execute that code.

In the case of a CnC specification, there are no such unnecessary constraints. This makes an analysis to remove these constraints unnecessary. Also an analysis cannot in general remove all unnecessary constraints, so with a CnC specification there can be more legitimate mappings to choose from, possibly resulting in an improved mapping.

Now compare static analysis of a CnC specification to static analysis of some form of explicit parallel code. Starting from parallel code, much of the work of the static analyzer is to understand the semantic constraints on parallel execution. But this is exactly what the CnC specification provides.

Compare a CnC application spec with some explicitly parallel representation of the application, where both have been tuned for a given parallel architecture. Assume the tuning and domain expert, who might be the same person but at different times, want to map the application to a distinct architecture. The constraints on the parallelism (the CnC spec) remain when the target architecture changes, whereas the explicit parallelism has to be rewritten. Starting with a CnC spec and tuning for one architecture facilitates the port to another architecture by isolating the constraints from the mapping. Only the tuning aspects change.

Now consider a scenario in which the domain expert and the tuning expert are distinct persons. It is possible that the tuning expert can do an adequate job by examining the specification and running some experiments. However, the two may have to cooperate for an effective mapping to the architecture. Their communication will not have much to do with the static aspects of the code within the steps. The questions that typically arise in this context are not about the step code but rather, for example, about typical sizes of the data instances in a collection, typical times of the computation steps in a collection, typical frequency that a step from a particular collection will actually produce or not produce data instances from a particular data collection, etc. The CnC spec serves as an interface between the two experts and provides the basis for such a discussion. The CnC spec, by hiding irrelevant information and providing a common language, raises the level of their discourse and facilitates their cooperation.

The goal of simplifying parallel programming led to the decision to make the language deterministic. Non-determinism and race conditions are the major source of difficulty in parallel programming. This decision has limited the scope of programs that can be

handled. On the other hand, CnC supports task, data, loop, pipeline, and tree parallelism in a clean uniform way. It also supports arbitrary scheduling approaches. The determinism of CnC disallows freedom from logical deadlocks, but since the deadlocks are deterministic, we can readily detect and correct them. Based on the dynamic single-assignment condition, program state is monotonic (see Section 4), which is a necessary condition for determinism.

A fundamental aspect of the design is the three kinds of collection constructs. We consider the rationale for this design choice in Section 3, after describing these constructs. Other design decisions related to specific features of CnC are also motivated in Section 3.

3. What is CnC?

Many of the concepts and constructs we discuss here are prevalent in the literature. For clarity, we will present CnC without presuming knowledge of related literature. We will discuss related work in Section 7.

The three main constructs in the CnC specification are *step collections*, *data collections*, and *control collections*. These are static collections. For each static collection, a set of dynamic *instances* is generated at runtime.

A step collection represents a high-level operation. High-level operations are partially ordered according to only their semantic constraints, i.e. data and control dependences. A data collection represents data that are consumed and produced by step collections. The production of a data collection by a step collection and its consumption by another step collection constitutes data dependence. A control collection is used to *prescribe* step instances, that is, to specify exactly which step instances will execute, which constitutes control dependence. Control collections are produced by step collections.

The CnC specification is represented as a graph where the nodes can be either step, data, or control collections. The edges represent *producer*, *consumer* and *prescription* (control) relations.

A whole CnC program includes the specification, the step code and the environment. The specification primarily specifies the semantic constraints of the application. The step code provides the implementation of the steps. The environment is the code that invokes the specification graph. The environment can produce and consume data and control collections.

The domain expert writes the CnC specification graph and the step implementation code and puts the steps together using the graph. In this sense CnC is a coordination language. In some problem domains, the above two tasks of the domain expert are generally performed by separate persons. The domain expert does not specify how operations are scheduled, which depends on the target architecture.

The tuning expert maps the CnC specification to an effective scheduling of that solution on a specific target architecture. In the case where operations are scheduled to execute on a parallel architecture, the domain expert in effect prepares the program for parallelism. This is quite different from the more common approach of embedding parallelism constructs within serial code.

The three fundamental constructs of the CnC model - data, control, and step collections - together form an interface that serves as a bridge between the domain and tuning experts. We now describe the three CnC collection constructs. Each control, data, or step instance is identified by a unique *tag*. These tags generally have meaning within the application but from the perspective of CnC they are simply arbitrary values that support a test for equality. The tag values may be composite, for example, they may be tuples of integers. Tags are used to parameterize instances of steps or of data. Tag types are from the serial language. The CnC runtime needs a way to manage them, e.g., hash them, but it does not care what types they are. The types are important to the step language. There is no reason CnC needs to constrain what types are used.

A *control collection* generalizes the iteration space construct. A named control collection creates a named iteration space, where instances of the collection correspond to points in the iteration space. Control collection C with tag i is denoted as $< C : i >$. The tag associated with an instance is analogous to an index. The instances in the space can not only be points in a grid, but also points in a recursion, in an irregular mesh, elements of a set, etc. The control collection mechanism supports control dependence, providing a way for the controller to determine points in this generalized iteration space which will control execution of some computation at some time in the future. There are several differences between control collections and iteration spaces:

- At a point in an ordered iteration space, the body of the controlled code is executed with the current index as a parameter. Next the body of code is executed with the next index as a parameter. In contrast, a control collection is an unordered set. The order that control tags are put into a control collection does not control the order in which the controlled step instances are executed. The execution of the controlled step instances in this sense is like the execution of a `forall` statement. A control tag dictates that the controlled code will be executed at some point with the tag as a parameter. So "if" and "when" are separated.
- An iteration space is over integer tuples. Typically these indices are used to access elements in an array. A control collection is over tags. The tag values may be used to refer not only to elements in an array but also to nodes in a tree, nodes in graph or, in general, elements in any set.
- An iteration space is used once and therefore unnamed. It is common to see multiple uses of the same iteration space in different parts of the code, but they must be specified separately. Further, a non-trivial analysis might be required to show that they are the same. A control collection may be used multiple times and is referred to by name. Object-oriented iterators also have this capability. That is, a control collection may control multiple computation step collections. It is clear without analysis that the control collections are identical for the multiple step collections.

A *data collection* creates a data space, where each instance in the collection is uniquely tagged. The unique tagging of data instances creates dynamic single assignment form, which supports determinism. In addition to the tag, each data instance has a value. A data collection named x is denoted $[x]$. An instance with tag value $[i, j]$ in data collection x is denoted as $[x : i, j]$. Due to dynamic single assignment, each instance is associated with exactly one value. The tag is analogous to a subscript, it parameterizes the data. Data collections support data dependence. The producer of a data collection need not know who the consumers are. Distinct data instances can be distributed (dynamically or statically) across time and space, and can be removed when no longer needed.

A *step collection* consumes data collections and produces data and control collections. Each step collection is controlled by a single control collection. There is a one-to-one correspondence between the instances of the step collection and the instances of this control collection. Each step instance has a tag and is controlled by a control instance with the same tag. A step collection named foo is denoted as (foo). Step instance i of step collection (foo) is denoted as $(foo : i)$. Where step collection (foo) consumes data collections $[x]$ and $[y]$, we denote this relationship as $[x], [y] - > (foo)$. A step collection may consume multiple instances of a data collection. Where step collection (foo) produces distinct data collections $[x]$ and $[y]$ and control collection $< C >$,

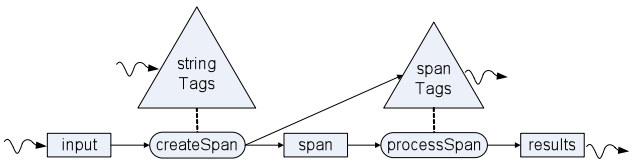

Figure 1. Graphical CnC Representation for an Example program

we denote this relationship as $(foo) - > [x], [y], < C >$. A step collection may produce multiple instances of a data collection or control collection. Each step collection is associated with a computation. Each step instance is associated with a computation that is parameterized by its tag. Distinct instances can be distributed (dynamically or statically) across time and space.

Control collections could be subsumed by data collections in that they both support dependences. But the existence of a data instance that would be consumed by a step if that step were to execute does not imply that the step will execute. Further, one cannot perform a get on a control collection.

A CnC specification can optionally include *tag functions* [3] and use them to specify the mapping between a step instance and the data instances that it *consumes* or *produces*. A tag function can be the identity function, or can define nearest neighbor computations, a parent/child in a tree, neighbors in a graph, or any other relationship useful in the application. We have chosen not to represent these via operators in the language for several reasons. Tags are not restricted to integers (they may well be domain-specific data types) and we anticipate that the set of interesting tag functions well evolve over time. We simply provide a mechanism for declaring tag functions. When tag functions are used to specify mappings from step instances to data instances in a CnC specification, the step code must conform to the data accesses specified. The functions are specified once and used by both the specification and the step code.

Examples of the use of tag functions in a CnC specification are shown below.

[x: parent(treeNodeID)] $-$ > (topDown: treeNodeID); // top-down processing gets parent

[y: NEWS(arrayLoc)] $-$ > (stencilCompuation: arrayLoc); // stencil computation gets North, South, East and West neighbors

An understanding of these functions and the resulting mappings provides a tuning expert with the information needed to tune the application, and provides a static analysis with the information needed to optimize distribution and scheduling of the application.

During execution, step, data, and control instances have state in the form of attributes. Instances acquire attributes monotonically during the execution. This simplifies program understanding; formulating and understanding the semantics; and is necessary for deterministic execution.

3.1 Example

We illustrate the process of creating a CnCgraph for a specific application. This discussion refers to Figure 1, which shows a simplified graphical representation of the application. The example is a trivial one. Both computation and data are too fine-grained to be realistic but the example illustrates many aspects of the model and is easy to follow. For example, its logic is representative of many data mining applications.

The program reads a set of strings. For each input string it generates a collection of strings that partitions the input into substrings that contain the same character. For example, if an instance of an input string contains "aaaffqqqmmmmmmmm", then it creates four substrings with values: "aaa", "ff", "qqq", "mmmmmmm". These instances are further processed as described below.

Step collections: The computation is partitioned into step collections, which are represented as ovals. In this application, there are two step collections: (createSpan) converts each string to a set of substrings. (processSpan) processes each of the substrings.

Data collections and producer-consumer relations: The data is partitioned into data collections, which are represented by rectangles in the graphical form. In this application there are three data collections, [input], [span] and [results]. These correspond to the input strings, the created substring spans and the results of processing these substring spans, respectively. The producer and consumer relationships between step collections and data collections are represented as directed edges between steps and data as shown in Figure 1.

Tags: Each dynamic step and data instance is uniquely identified by a tag. A tag might be composed of several components, for example, employee number and year or maybe xAxis, yAxis, and iterationNum. In our example, the instances of the [input] data collection are distinguished by stringID. The (createSpan) step instances are also distinguished by stringID. The instances of the [span] data collection, the (processSpan) step collection, and the [results] data collection are distinguished by both a stringID and a spanID within the string.

Control Collections and Prescriptions: Distinguishing instances of (createSpan) steps by values of stringID does not tell us if a (createSpan) step is to be executed for a particular stringID, such as stringID 58. This is the role of control collections. Control collections are shown as triangles. There are two control collections in our example graph. A control collection in $< stringTags >$ identifies the set of strings. A control collection in $< spanTags >$ identifies the set of substrings. A prescriptive relation may exist between a control collection, ($< stringTags >$ for example) and a step collection ((createSpan) for example). The meaning of such a relationship is this: if a control instance t, say stringID 58, is in $< stringTags >$, then the step instance s in (createSpan) with tag value stringID 58, will execute. The prescription relation mechanism determines if a step instance will execute. When it executes is up to a subsequent scheduler. A prescriptive relation is shown as a dotted edge between a control collection and a step collection. The form of the tags for step instances is identical to the form of the tags of its prescribing control instances. The instances of the control collection $< stringTags >$ and of the step collection (createSpan) are both distinguished by stringID. The work of (processSpan) step instances is to perform some processing on each of the substrings generated. A (processSpan) step instance will execute for each substring. The control collection $< spanTags >$ identifies a stringID and a spanID for each substring to be processed. $< spanTags >$ tags will prescribe (processSpan) step instances. Now we consider how the control collections are produced. The control instances in $< stringTags >$ are produced by the environment, which also produces data instances. The step collection (createSpan) not only produces data instances in [span] but also produces the tags in $< tagSpan >$ that identify them. In this example, the corresponding instances in the collections of $< spanTags >$, [span] and (processSpan) have the same tags.

Environment The environment (the code that invokes the graph) can produce and consume data and control collections. These relationships are represented by directed squiggly edges. In our application, for example, `env` produces [input] data and consumes [results] data.

This example illustrates some of the basic capabilities of CnC but it is simplified in several ways. CnC handles graphs with cycles and more interesting tag functions. In addition to specifying the graph, we need to code the steps and the environment in a serial

language. The step has access to its tag values. It uses get operations to consume data instances and put operations to produce data and control instances.

4. Formal Semantics

In this section we describe the semantics of a Featherweight CnC language. Using these semantics we can show that the language is provably deterministic. We begin by describing the syntax and semantics of the Featherweight CnC language and then discuss its relation to the full CnC language.

4.1 Syntax

The grammar for Featherweight CnC is shown in Figure 2. We provide a syntax that combines the language for specifying data and step collections along with the base language for writing the step bodies. Combining the languages into one grammar allows us to give a complete semantics for the execution of a Featherweight CnC program.

A Featherweight CnC program is of the form

```
f1(int a) {d1 s1}
f2(int a) {d2 s2}
...
fn(int a) {dn sn}
```

where each f_i is a step, the a is the tag passed to a step instance, and the d_i and s_i make up the step body. The d_i are the initial gets for the data needed by the step and the s_i are the statements in the step bodies. Writing a data item to a data collection is represented by the $data.put(e1, e2)$ production. The expression $e1$ is used to compute the tag and the expression $e2$ is used to compute the value. Tag collections have been removed in favor of directly starting a new step using the $prescribe$ statement. A computation step consists of a series of zero or more declarations followed by one or more statements. These declarations introduce names local to the step and correspond to performing a get on a data collection. A step must finish all of its gets before beginning any computation. The computation in a step is limited to simple arithmetic, writing to a data collection, and starting new computation steps.

$$
\begin{array}{lrcll}
Program: & p & ::= & f_i(int\ a)\{d_i\ s_i\}, i \in 1..n & \\
Declaration: & d & ::= & n = data.get(e); d & \\
 & & | & \epsilon & \\
Statement: & s & ::= & skip & \\
 & & | & if\ (e > 0)\ s_1\ else\ s_2 & \\
 & & | & data.put(e_1, e_2); s & \\
 & & | & prescribe\ f_i(e); s & \\
Expression: & e & ::= & c & \text{(integer constant)} \\
 & & | & n & \text{(local name)} \\
 & & | & a & \text{(formal parameter name)} \\
 & & | & e_1 + e_2 &
\end{array}
$$

Figure 2. The grammar of Featherweight CnC. We use c to range over integer constants, and n to range over variable names.

4.2 Semantics

We use A to denote the state of the array $data$, that is, a total mapping from indices (which are integers) to either integers or $\perp$ (which denotes undefined). We use A_0 to denote the mapping that maps every index to $\perp$. We define an ordering $\sqsubseteq$ on array mappings such that $A \sqsubseteq A'$ if and only if $dom(A) \subseteq dom(A')$ and for all $n \in dom(A) \cap dom(A) : A(n) = A'(n)$.

For an expression e in which no names occur, we use $[\![e]\!]$ to denote the result to which e evaluates.

We will now define a small-step operational semantics for the language. Our main semantic structure is a tree defined by the following grammar:

$$Tree: \quad T \quad ::= \quad T \parallel T \quad | \quad (d\ s)$$

We assert that $\parallel$ is associative and commutative, that is

$$
\begin{aligned}
T_1 \parallel (T_2 \parallel T_3) &= (T_1 \parallel T_2) \parallel T_3 \\
T_1 \parallel T_2 &= T_2 \parallel T_1
\end{aligned}
$$

A state in the semantics is a pair (A, T) or error. We use σ to range over states. We define an ordering $\leq$ on states such that $\sigma \leq \sigma'$ if and only if either $\sigma' =$ error, or if $\sigma = (A, T)$ and $\sigma' = (A', T)$, then $A \sqsubseteq A'$.

We will define the semantics via a binary relation on states, written $\sigma \rightarrow \sigma'$. The initial state of an execution of s is (A_0, s). A final state of the semantics is of either of the form $(A, skip)$, of the form error, or of the form (A, T) in which every occurrence in T of $(d\ s)$ has that property that d is of the form $n = data.get(e); d'$ and $A[\![e]\!] = \perp$.

We now show the rules that define $\rightarrow$.

$$(A, skip \parallel T_2) \rightarrow (A, T_2) \tag{1}$$

$$(A, T_1 \parallel skip) \rightarrow (A, T_1) \tag{2}$$

$$\frac{(A, T_1) \rightarrow \text{error}}{(A, T_1 \parallel T_2) \rightarrow \text{error}} \tag{3}$$

$$\frac{(A, T_2) \rightarrow \text{error}}{(A, T_1 \parallel T_2) \rightarrow \text{error}} \tag{4}$$

$$\frac{(A, T_1) \rightarrow (A', T_1')}{(A, T_1 \parallel T_2) \rightarrow (A', T_1' \parallel T_2)} \tag{5}$$

$$\frac{(A, T_2) \rightarrow (A', T_2')}{(A, T_1 \parallel T_2) \rightarrow (A', T_1 \parallel T_2')} \tag{6}$$

$$(A, if\ (e > 0)\ s_1\ else\ s_2) \rightarrow (A, s_1) \quad (\text{if } [\![e]\!] > 0) \tag{7}$$

$$(A, if\ (e > 0)\ s_1\ else\ s_2) \rightarrow (A, s_2) \quad (\text{if } [\![e]\!] \leq 0) \tag{8}$$

$$(A, data.put(e_1, e_2); s) \rightarrow (A[[\![e_1]\!] := [\![e_2]\!]], s) \quad (\text{if } A[[\![e_1]\!]] = \perp) \tag{9}$$

$$(A, data.put(e_1, e_2); s) \rightarrow \text{error} \quad (\text{if } A[[\![e_1]\!]] \neq \perp) \tag{10}$$

$$(A, prescribe\ f_i(e); s) \rightarrow (A, ((d_i\ s_i)[a := [\![e]\!]]) \parallel s) \quad (\text{the body of } f_i \text{ is } (d_i\ s_i)) \tag{11}$$

$$(A, n = data.get(e); d\ s) \rightarrow (A, (d\ s)[n := [\![e]\!]]) \quad (\text{if } A[[\![e]\!]] \neq \perp) \tag{12}$$

Notice the side condition of Rules (9, 10) is the single-assignment condition and the side condition of Rule (12) is the use-after-assignment condition.

4.3 Discussion

Given the semantics above we formally state the desired property of determinism as Theorem 1 and we have proved this theorem for Featherweight CnC. The key language feature that enables determinism is the single assignment condition. It is the single assignment condition that guarantees the monotonicity of the data collection A. We view A as a partial function from integers to integers and the single assignment condition guarantees that we can establish an ordering based on the non-decreasing domain of A. The monotonic behavior of a Featherweight CnC program is an important part of the determinism proof. The supplemental document contains the full proof of Theorem 1.

Theorem 1. (Determinism) *If* $\sigma \rightarrow^* \sigma'$ *and* $\sigma \rightarrow^* \sigma''$*, and* σ', σ'' *are both final states, then* $\sigma' = \sigma''$*.*

The Featherweight CnC language presented here captures the essence of the full CnC programming model. Three key features of CnC are represented directly in the semantics for Featherweight CnC. First, the *single assignment* property that only allows one write to a data collection for a given tag. This property shows up as the side condition of Rules (9,10). Second, the *data dependence* property that says a step cannot execute until all of the data it needs is available. This property shows up as the side condition of Rule (12). Third, the *control dependence* property that separates *if* a step will execute from *when* a step will execute. This property is captured by Rule (11) which shows that when a step is prescribed it will be added to the set of steps that will be executed, but says nothing about when the step will actually execute. Finally, the feature of CnC that restricts the body of a step to perform all of the get operations before doing any put operations is encoded directly in the grammar of the language.

Although there are some features of CnC that are missing from Featherweight CnC, we can extend Featherweight CnC to include the higher level features. The main features that are missing from Featherweight CnC are named data and tag collections. Named data collections can be added by simply allowing data collections other than the single $data$ array in Featherweight CnC. Named tag collections could be added with a few more rules for the grammar and semantics. A put on a tag collection would correspond to a $prescribe$ call to the function passing the tag as a parameter. With these extensions we could represent the full CnC language. However, these additions do not add any power to the CnC model captured by Featherweight CnC.

In the Featherweight CnC language we have combined the language for writing step bodies with the coordination language for coordinating the step computations. Real CnC programs will be written using a full featured programming language such as C or Java. We should then ask whether we have lost some computational power by having such a simple step language. We argue that the language provided for writing step bodies is powerful enough to encode the set of all while programs, which are known to be Turing complete. While programs have a very simple grammar consisting of a while loop, a single variable x, assigning zero to x, and incrementing x by one. We can write a translator that will convert a while program to a Featherweight CnC program by using recursive $prescribe$ statements to encode the while loop. The value of x can be tracked by explicitly writing the value to a new location in the $data$ array at each step and passing the tag for the current location of x to the next step. We argue that the Featherweight CnC language is Turing complete and therefore we have not lost any computational power.

The underlying step language could cause problems for determinism if it violates the rules of the semantics. In particular, the data collection in the semantics maps integers to integers. Once a data item is inserted into the collection, it cannot be changed. If we are writing the steps in C with a data collection that contains pointers, then we could get non-deterministic behavior by modifying the data pointed to by the data collection after it has been inserted. As long as the data collection contains values or the data pointed to is not modified after inserting into the collection, the determinism property should extend to a general purpose step language.

We do not claim any freedom from deadlocks in Featherweight CnC. We can see from the semantics that a final state can be one in which there are still steps ready to run, but none can make progress because the required data is unavailable. We say the computation has reached a *quiescent* state when this happens. Deadlock is only a problem when it happens non-deterministically because it makes errors difficult to detect and correct. We have proved that Featherweight CnC is deterministic so any computation that reaches a quiescent final state will always reach that same final state. In this setup deadlock is not a problem because we can readily detect and correct the issue.

5. Mapping CnC to Target Platforms

Implementations of CnC need to provide a translator and a runtime. The translator maps the CnC syntax to the runtime API in the target language. One of the attractive properties of the CnC model is that it has a small number of constructs, properties and operations. These need to be mapped to the implementation language and platform. Table 1 shows the mappings for three different CnC implementations.

In dynamic runtime systems, we influence the mappings through alternative scheduling strategies, such as LIFO, FIFO or priority-based. In static runtime systems, implementer provides the mappings.

The way an application is mapped will determine the execution time, memory footprint, power, latency, bandwidth etc. Any/all of these can be used as a criteria for the mapping. The resulting execution might exhibit data, pipeline, or task parallelism, or some combination of these. The kind of parallelism to be exploited is not a part of the CnCspecification.

5.1 Implementing CnC on Different Platforms

Table 1 shows how different CnC constructs and actions are implemented on three different platforms: Intel's Threading Building Blocks (TBB), Java Concurrency Utilities (JUC) and the .NET Task Parallel Library (TPL). While most of the entries in this table are self-explanatory, some of them warrant further discussion.

In all implementations, step prescription involves creation of an internal data structure representing the step to be executed. In the .NET implementation, a parallel task to execute the step will be spawned eagerly upon prescription, while in the TBB and JUC implementations, the task spawning could be delayed (depending on the policy) until the data needed by the task is ready.

The get operations on a data collection could be blocking (in cases when the task executing the step could be spawned before all the inputs for the step are available) or non-blocking (the runtime guarantees that the data is available when get is executed) in the JUC implementation. The .NET implementation always blocks the task executing the get if the data is not yet available in the data collection. The TBB implementation always uses a non-blocking get with roll back and replay described below.

Both the TBB and JUC implementations have a roll back and replay policy, which aborts the step performing a get on an unavailable data item and puts the step in a separate list associated with the failed get. When a corresponding put of the item gets executed, all the steps in a list waiting on that item are restarted.

JUC implementation also has a delayed async policy, which requires the user or the compiler to provide a boolean ready() method that evaluates to true once all the inputs required by the step are available. Only when ready() for a given step evaluates to true does the JUC implementation spawn a task to execute the step.

All implementations require some code for initialization of the CnC graph, which involves creation of step objects, creation of a graph object, connecting the concrete implementations of the steps to the abstract step declarations from the CnC program and performing the initial puts into the data and control collections.

In the TBB implementation, ensuring that all the steps in the graph have finished execution is done by calling the run() method on the graph object. This method blocks until all the steps in the program have completed. This is implemented differently by different schedulers, for example in the queue based schedulers

	Intel Threading Building Blocks	Java Concurrency Utilities	.NET Task Parallel Library
Step Language	C, C++	Java, HJ	C#, F#, VB.NET
Scheduler	TBB scheduler, custom CnC scheduler	HJ work-sharing scheduler on j.u.c. ThreadPoolExecutor	TPL scheduler
Step Prescription	Creation of a custom closure	Creation of a custom closure	TPL task creation
Step Execution	TBB task creation: Eager (when prescribed), Delayed (when ready)	HJ async call: Eager (when prescribed), Delayed (when ready)	TPL task creation: Eager (when prescribed)
Data Structure for Data Collections	TBB::CHM (concurrent hashmap)	JUC ConcurrentHashMap	Custom concurrent hash table based on .NET Dictionary + ReadWriteLock
Basic *put* and *get* operations	Non-blocking *puts* and *gets*	Non-blocking *puts*, blocking *gets* *gets* with coarse-grained or fine-grained wait/notify	Non-blocking *puts* and blocking *gets* with fine-grained wait/notify
Optimized *put* and *get*	Roll back & replay	Nonblocking *puts* and *gets* with delayed async's, roll back & replay	None
Data and Control Tags	Any type that supports equality comparison	Value types: Java strings, HJ Points	Immutable types: F# Tuples
CnC Graph Termination	Graph.run() (scheduler specific)	HJ finish operation	CountdownEvent (atomic counter)
Safety properties	Single Assignment Rule	CnC Graph Conformance, Single Assignment Rule, Tag Immutability	CnC Graph Conformance, Single Assignment Rule, Tag Immutability

Table 1. Comparison of Three CnC Implementations

when the number of workers waiting to execute a task reaches the total number of workers then there are no more steps to execute.

In the JUC implementation, ensuring that all the steps in the graph have completed is done by enclosing all the control collection puts from the environment in an HJ finish construct. The finish construct [16] ensures that all transitively spawned tasks (which includes all the tasks for step execution) have completed.

In addition to the differences between step implementation languages, different CnC implementations approach enforcing the CnC graph properties differently. The TBB implementation performs runtime checks of the single assignment rule, while the JUC and .NET implementations also check for CnC conformity (e.g., a step cannot perform a put into an data collection if that relationship is not specified in the CnC graph), and ensure tag immutability.

Another aspect of CnC runtimes is garbage collection. Unless the runtime at some point deletes the items that were put, the memory usage will continue to increase. The managed runtimes such as Java or .NET do not help here, since the data instances are still referenced in the data store. [3] introduces a declarative *slicing annotation* for CnC that can be transformed into a reference counting procedure for memory management.

CnC *C++ Implementation* The C++ runtime implements a task-based model. Step instances are queued inside the scheduler as tasks, which are picked up by the workers in its thread pool.

The C++ runtime is provided as a template library, allowing tags and data instances to be of any type and type checking at compile time. Templates also provide optimization opportunities to the compiler.

The C++ runtime provides several schedulers, most of which are based on the TBB schedulers, which use work stealing [2, 9]. Differences in scheduling reflect in the way the steps are handed over to the TBB scheduler. The default TBB scheduler uses tbb::parallel_while, which allows the creation of child steps. Another scheduler uses tbb::parallel_for. Instead of creating child steps, this scheduler performs breadth-first processing.

Habanero-Java CnC *Implementation* The Java-based Habanero-Java (HJ) programming language is an extension to v0.41 of the X10 language described in [6]. Since HJ is based on Java, the use of certain primitives from the Java Concurrency Utilities [22] is also permitted in HJ programs, most notably operations on Java Concurrent Collections such as `ConcurrentHashMap` from `java.util.concurrent`, and on Java *atomic* variables. The mapping of CnC primitives to HJ is summarized in Table 1. For a detailed description of the runtime mapping and the code translator from CnC to HJ , see the supplemental document.

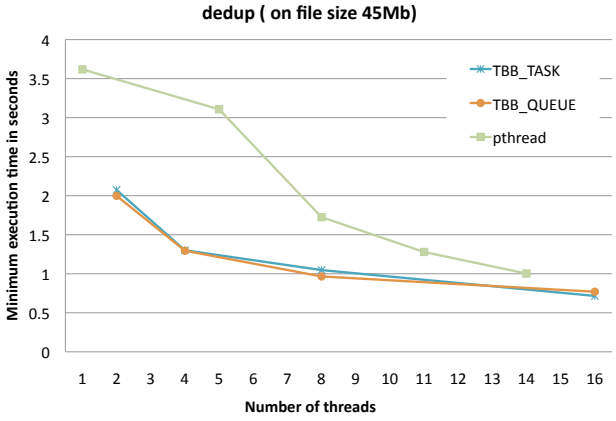

Figure 3. Execution time for pthreads and CnC-TBB implementations of the PARSEC dedup benchmark with 45MB input size on a 16-core Intel Caneland SMP, as a function of number of worker threads used.

.NET Implementation of CnC The prototype CnC.NET implementation takes full advantage of language generics to implement type-safe *put* and *get* operations on item and tag collections. The runtime and code generator are written in F#, but because of the strong language interoperability of the .NET platform, the step bodies can be written in any language that runs on the CLR (Common Language Runtime), including F#, C#, and VB.NET.

6. Experimental Results

In this section, we present experimental results obtained using the TBB and JUC implementations of CnC outlined in the previous section. The final version of the paper will include results from the .NET implementation as well.

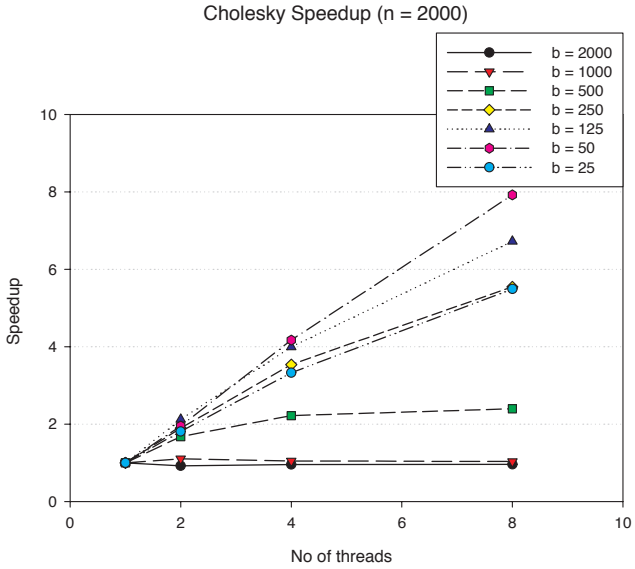

Figure 4. Speedup results for TBB implementation of 2000×2000 Cholesky Factorization CnC program on an 8-way (2p4c) Intel® dual Xeon Harpertown SMP system. The running time for the baseline (1 thread, tile size 2000) was 24.911 seconds

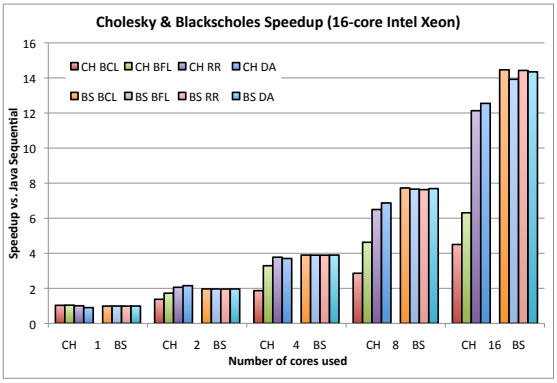

Figure 5. Speedup results for CnC-Java implementation of 2000×2000 Cholesky Factorization CnC program with tile size 100 and Black-Scholes CnC program on a 16-way (4p4c) Intel® Xeon SMP system.

6.1 TBB Implementation

We have ported Dedup, a benchmark from the PARSEC [1] benchmark suite, to the TBB implementation of CnC with steps written in C/C++. Figure 3 shows the execution time for pthreads and CnC-TBB implementations of the dedup benchmark with 45MB input size on a 16-core Intel Caneland 3GHz SMP with 8GB memory, as a function of the number of worker threads used. The pthreads version is the standard version obtained from the PARSEC site.

Figure 3 shows that CnC has superior performance to the pthreads implementation of Dedup. With a fixed number of threads per stage, load imbalance between the stages limits the parallelism in the pthreads implementation, while the CnC implementation does not have this issue, because all threads can work on all stages. In addition to improved performance, the CnC version also simplifies the work of the programmer, who simply performs *put*s and *get*s without the need to think about lower-level parallelism

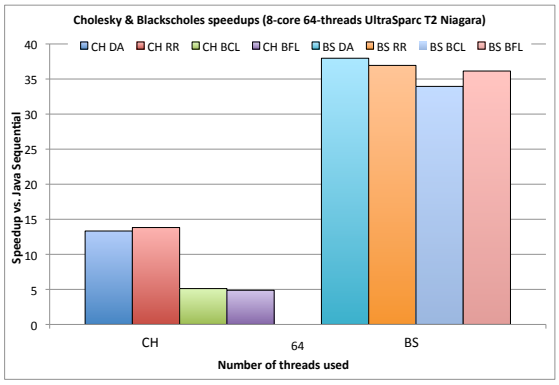

Figure 6. Speedup results for CnC-Java implementation of 2000×2000 Cholesky Factorization CnC program with tile size 80 and Black-Scholes CnC program on a 8 core 64 thread UltraSPARC T2 Sun Niagara system.

mechanisms such as explicit threads, mutexes and conditional variables. We have experimented with two CnC scheduling policies: TBB_TASK and TBB_QUEUE. TBB_TASK wraps a step instance in a tbb::task object and eagerly spawns it, thereby delegating the scheduling to the TBB scheduler without much overhead. TBB_QUEUE provides a global FIFO task queue, populated with scheduled steps, who are consumed by multiple threads. This policy is a good match for a pipelined algorithm such as dedup.

The second CnC TBB example that we evaluated was a Cholesky Factorization of size 2000×2000 . This is an interesting example because its performance is impacted by both parallelism (number of cores used) and locality (tile size), thereby illustrating how these properties can be taken into account when tuning a CnC program. Figure 4 shows the speedup relative to a sequential version on an 8-way Intel® dual Xeon Harpertown SMP system. A deeper study of the CnC TBB Cholesky example can be found in an upcoming poster abstract [7].

We have also tested CnC microbenchmarks such as N-Queens and the Game of Life and compared them to their TBB and OpenMP implementations. We omit the full results due to space constraints, but CnC implementation matched the scalability of the TBB and OpenMP implementations, while providing a much easier to use, higher level programming model.

6.2 Java Implementation

Figure 5 shows the speedup of the HJ CnC Cholesky factorization [7] and Blackscholes [1] implementations on a 16-way Intel Xeon SMP, using four different strategies for data synchronization when performing *put*s and *get*s on the item collection: blocked using coarse-grain locking of the whole item collection (CH-BCL and BS-BCL), fine-grain locking on individual data items in the item collection (CH-BFL and BS-BFL), data-driven computation using a roll-back and replay (CH-RR and BS-RR), and a non-blocking strategy using the delayed async (CH-DA and BS-DA). The speedups reported on Figure 5 are relative to the sequential Cholesky and Blackscholes implementations. For Cholesky, we can observe that the CnC infrastructure adds moderate overhead (28%-41%, depending on the synchronization strategy) in a single thread case, while Blackscholes only shows minimal (1%-2%) overhead. Data driven and non-blocking strategy scale very well, while we can see the negative effects of the fine-grain locking and especially coarse-grain locking strategy after 8 processors. For Blackscholes, since all of the data is available to begin with and we can evenly par-

tition the data, we do not see much difference between the different implementation strategies. We obtained the measurements for data points below 16 processors by limiting the number of processors used for execution in the operating system (Linux).

Figure 6 shows the speedup for the 64-thread case for Cholesky and Blackscholes on the UltraSPARC T2 (8 cores with 8 threads each). We see very good scaling for Blackscholes using 64-threads but we can start to see the negative effects of coarse-grain locking for both applications. Even though the data is partitioned among the workers, the coarse-grain locking causes contention on the single lock which results in worse scalability than the fine-grained and non-blocking versions. Cholesky, which is mostly floating point computation, achieves a better than 8x speedup for 64 threads, which is a very reasonable result considering that the machine has only 8 floating point units. We have not been able to restrict the number of threads on Solaris using the work-sharing HJ runtime the way we have been able to do so on Linux.

7. Related Work

Parallel prog. model	Declarative	Deterministic	Efficient
Intel TBB [23]	No	No	Yes
.Net Task Par. Lib. [27]	No	No	Yes
Cilk [11]	No	No	Yes
OpenMP [5]	No	No	Yes
CUDA [21]	No	No	Yes
Java Concurrency [22]	No	No	Yes
Det. Parallel Java [24]	No	Hybrid	Yes
High Perf. Fortran [19]	Hybrid	No	Yes
X10 [6]	Hybrid	No	Yes
Linda [12, 13]	Hybrid	No	Yes
Asynch. Seq. Processes [4]	Yes	Yes	No
StreamIt [14, 15]	Yes	Yes	Yes
LabVIEW [28]	Yes	Yes	Yes
CnC [this paper]	Yes	Yes	Yes

Table 2. Comparison of several parallel programming models.

We use Table 2 to guide the discussion in this section. This table classifies programming models according to their attributes in three dimensions: *Declarative*, *Deterministic* and *Efficient*. Space limitations prevent us from listing all known parallel programming models due to the vast amount of past work in this area. Instead, we include a few representative examples for each distinct set of attributes, and trust that the reader can extrapolate this discussion to other programming models with similar attributes in these three dimensions.

A number of low-level programming models in use today — *e.g.,* Intel TBB [23], .Net Task Parallel Library [27], Cilk [11], OpenMP [5], CUDA [21], Java Concurrency [22] — are non-declarative, non-deterministic, and efficient[1]. Deterministic Parallel Java [24] is an interesting variant of Java; it includes a subset that is provably deterministic, as well as constructs that explicitly indicate when determinism cannot be guaranteed for certain code regions, which is why it contains a "hybrid" entry in the *Deterministic* column. The next three languages in the table — High Performance Fortran (HPF) [19], X10 [6], Linda [12, 13] — contain hybrid combinations of imperative and declarative programming in different ways. HPF combines a declarative language for data distribution and data parallelism with imperative (procedural) statements, X10 contains a functional subset that supports declarative parallelism, and Linda is a coordination language in which a thread's interactions with the tuple space is declarative.

[1] We call a programming model efficient if there are known implementations that can deliver competitive performance for a reasonably broad set of programs in that model.

Linda deserves further attention in this discussion since it was a major influence on the CnC design. CnC shares two important properties with Linda: both are coordination languages that specify computations and communications via a tuple/tag namespace, and both create new computations by adding new tuples/tags to the namespace. However, CnC differs from Linda in many ways. For example, an in() operation in Linda atomically removes the tuple from the tuple space, but a CnC get() operation does not remove the item from the collection. This is a key reason why Linda programs can be non-deterministic in general, and why CnC programs are provably deterministic. Further, there is no separation between tags and values in a Linda tuple; instead, the choice of tag is implicit in the use of wildcards. In CnC, there is a separation between tags and values, and control tags are first class constructs like data items.

The last four programming models in the table are both declarative and deterministic. Asynchronous Sequential Processes [4] is a recent model with a clean semantics, but without any efficient implementations. In contrast, the remaining three entries are efficient as well. StreamIt [14, 15] is a representative of a modern streaming language, and LabVIEW [28] is a representative of a modern dataflow language. Both streaming and dataflow languages have also had major influences on the CnC design.

The CnC semantic model is based on dataflow in that steps are functional and the execution only needs to wait for data becoming ready thereby ensuring that there's no extra serialization. However, CnC differs from dataflow in some key ways. The use of control tags elevates control to a first-class construct in CnC. In addition, item collections allow more general indexing (as in a tuple space) compared to dataflow arrays (I-structures).

CnC is like streaming in that the internals of a step are not visible from the graph that describes their connectivity, thereby establishing an isolation among steps. A producer step in a streaming model need not know its consumers; it just needs to know which buffers (collections) to perform read and write operations on. However, CnC differs from streaming in that put() and get() operations need not be performed in a FIFO order, and (as mentioned above) control is a first-class construct in CnC.

For completeness, we also include a brief comparison with graphical coordination languages in a distributed system, using Dryad [18] as an exemplar in that space. Dryad is a general-purpose distributed execution engine for coarse-grain data-parallel applications. It combines sequential vertices with directed communication channels to form an acyclic dataflow graph. Channels contain full support for distributed systems and are implemented using TCP, files, or shared memory pipes as appropriate. The Dryad graph is specified by an embedded language (in C++) using a combination of operator overloading and API calls. The main difference with CnC is that CnC can support cyclic graphs with first-class tagged controller-controllee relationships and tagged item collections. Also, the CnC implementations described in this paper are focused on multicore rather than distributed systems

In summary, CnC has benefited from influences in past work, but we're not aware of any other parallel programming model that shares CnC's fundamental properties as a coordination language, a declarative language, a deterministic language, and as a language that is amenable to efficient implementations.

8. Conclusions

This paper presents a programming model for parallel computation. A computation is written as a CnC graph, which is a high-level, declarative representation of semantic constraints. This representation can serve as a bridge between domain and tuning experts, facilitating their communication by hiding information that is not relevant to both parties. We prove deterministic execution of the model. Deterministic execution simplifies program analysis

and understanding, and reduces the complexity of compiler optimization, testing, debugging, and tuning for parallel architectures. We also present a set of experiments that show several implementations of CnC with distinct base languages and distinct runtime systems. The experiments confirm that the CnC model can express and exploit a variety of types of parallelism at runtime. When compared to the state of the art low-level parallel programming models, our experiments establish that the CnC programming model implementations deliver competitive raw performance, equal or better scalability, and superior programmability and ease of use.

References

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work-stealing. In *Proceedins of the 35th Annual IEEE Conference on Foundations of Computer Science*, 1994.

[3] Zoran Budimlić, Aparna M. Chandramowlishwaran, Kathleen Knobe, Geoff N. Lowney, Vivek Sarkar, and Leo Treggiari. Declarative aspects of memory management in the concurrent collections parallel programming model. In *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 47–58, New York, NY, USA, 2008. ACM.

[4] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous sequential processes. *Information and Computation*, 207(4):459–495, 2009.

[5] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Programming in OpenMP*. Academic Press, 2001.

[6] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vivek Sarkar, and Christoph Von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538. ACM SIGPLAN, 2005.

[7] Anonymized for double-blind submission.

[8] Intel Corporation. Threading Building Blocks. http://www.threadingbuildingblocks.org/.

[9] Intel Corporation. Intel(R) Threading Building Blocks reference manual. Document Number 315415-002US, 2009.

[10] R.Barik et al. Experiences with an smp implementation for x10 based on the java concurrency utilities. In *Workshop on Programming Models for Ubiquitous Parallelism (PMUP), held in conjunction with PACT 2006*, September 2006.

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of PLDI'98, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[12] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[13] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[14] M. I. Gordon et al. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM.

[15] M. I. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[16] Habanero multicore software research project web page. http://habanero.rice.edu.

[17] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. Mcrt-malloc – a scalable transactional memory allocator. In *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, pages 74–83, June 2006.

[18] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[19] Ken Kennedy, Charles Koelbel, and Hans P. Zima. The rise and fall of High Performance Fortran. In *Proceedings of HOPL'07, Third ACM SIGPLAN History of Programming Languages Conference*, pages 1–22, 2007.

[20] Kathleen Knobe and Carl D. Offner. Tstreams: A model of parallel computation (preliminary report). Technical Report HPL-2004-78, HP Labs, 2004.

[21] NVIDIA. CUDA reference manual. Version 2.3, 2009.

[22] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[23] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[24] Jr. Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for Deterministic Parallel Java. In *Proceedings of OOPSLA'09, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 97–116, 2009.

[25] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.

[26] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.

[27] Stephen Toub. Parallel programming and the .NET Framework 4.0. http://blogs.msdn.com/pfxteam/archive/2008/10/10/8994927.aspx, 2008.

[28] Jeffrey Travis and Jim Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun*. Prentice Hall, 2006. 3rd Edition.

[29] X10 v1.7 language specification. http://x10.sourceforge.net/docs/x10-170.pdf.

A. Proof of Determinism

Lemma 1. (Error propagation)

1. *If* $(A, T_1) \rightarrow^* \mathsf{error}$*, then* $(A, T_1 \parallel T_2) \rightarrow^* \mathsf{error}$.
2. *If* $(A, T_2) \rightarrow^* \mathsf{error}$*, then* $(A, T_1 \parallel T_2) \rightarrow^* \mathsf{error}$.

Proof. Let us first prove item 1. It is sufficient to prove

> For all $n \geq 0$: if $(A, T_1) \rightarrow^n \mathsf{error}$, then $(A, T_1 \parallel T_2) \rightarrow^* \mathsf{error}$.

We will prove that by induction on n.

For $n = 0$, we have that $(A, T_1) \rightarrow^0 \mathsf{error}$ is impossible. For $n = 1$, the property follows from Rule (3).

In the induction step, suppose the property holds for n, where $n \geq 1$, and suppose $(A, T_1) \rightarrow^{n+1} \mathsf{error}$. This sequence of steps must be of the form: $(A, T_1) \rightarrow (A', T_1') \rightarrow^n \mathsf{error}$. From $(A, T_1) \rightarrow (A', T_1')$ and Rule (5), we have $(A, T_1 \parallel T_2) \rightarrow (A', T_1' \parallel T_2)$. From $(A', T_1') \rightarrow^n \mathsf{error}$ and the induction hypothesis, we have $(A', T_1' \parallel T_2) \rightarrow^* \mathsf{error}$. We conclude $(A, T_1 \parallel T_2) \rightarrow^* \mathsf{error}$.

The proof of item 2 is similar to the proof of item 1; we omit the details. □

Lemma 2. (Structural multistep)

1. *If* $(A, T_1) \rightarrow^* (A', T_1')$*, then* $(A, T_1 \parallel T_2) \rightarrow^* (A', T_1' \parallel T_2)$.
2. *If* $(A, T_2) \rightarrow^* (A', T_2')$*, then* $(A, T_1 \parallel T_2) \rightarrow^* (A', T_1 \parallel T_2')$.

Proof. Let us first prove item 1. It is sufficient to prove

> For all $n \geq 0$: if $(A, T_1) \rightarrow^n (A', T_1')$, then $(A, T_1 \parallel T_2) \rightarrow^* (A', T_1' \parallel T_2)$.

We will prove that by induction on n.

For $n = 0$, we have that $(A, T_1) = (A', T_1')$ so $(A, T_1 \parallel T_2) = (A', T_1' \parallel T_2)$.

In the induction step, suppose the property holds for n, where $n \geq 0$, and suppose $(A, T_1) \rightarrow^{n+1} (A', T_1')$. This sequence of steps must be of the form: $(A, T_1) \rightarrow (A'', T_1'') \rightarrow^n (A', T_1')$. From $(A, T_1) \rightarrow (A'', T_1'')$ and Rule (5), we have $(A, T_1 \parallel T_2) \rightarrow (A'', T_1'' \parallel T_2)$. From $(A'', T_1'') \rightarrow^n (A', T_1')$ and the induction hypothesis, we have $(A'', T_1'' \parallel T_2) \rightarrow^* (A', T_1' \parallel T_2)$. We conclude $(A, T_1 \parallel T_2) \rightarrow^* (A', T_1' \parallel T_2)$.

The proof of item 2 is similar to the proof of item 1; we omit the details. □

Lemma 3. (Error Preservation) *If* $(A, T) \rightarrow \mathsf{error}$ *and* $A \sqsubseteq A'$*, then* $(A', T) \rightarrow \mathsf{error}$.

Proof. Straightforward by induction on the derivation of $(A, T) \rightarrow \mathsf{error}$; we omit the details. □

Lemma 4. (Monotonicity) *If* $\sigma \rightarrow \sigma'$*, then* $\sigma \leq \sigma'$.

Proof. Straightforward by induction on the derivation of $\sigma \rightarrow \sigma'$. The interesting case is for Rule (9) which is where σ can change and the single-assignment side-condition plays an essential role. We omit the details. □

Lemma 5. (Clash) *If* $(A, T) \rightarrow (A', T')$ *and* $A[c] = \bot$ *and* $A'[c] \neq \bot$ *and* $A_d[c] \neq \bot$ *and then* $(A_d, T) \rightarrow \mathsf{error}$.

Proof. Straightforward by induction on the derivation of $(A, T) \rightarrow (A', T')$; we omit the details. □

Lemma 6. (Independence) *If* $(A, T) \rightarrow (A', T')$ *and* $A'[c] = \bot$*, then* $(A[c := c'], T) \rightarrow (A'[c := c'], T')$. ,

Proof. From $(A, T) \rightarrow (A', T')$ and Lemma 4 we have $A \sqsubseteq A'$. From $A \sqsubseteq A'$ and and $A'[c] = \bot$, we have $A[c] = \bot$. The proof is now straightforward by induction on the derivation of $(A, T) \rightarrow (A', T')$; we omit the details. □

Lemma 7. (Diamond) *If* $(A, T_a) \rightarrow (A', T_a')$ *and* $(A, T_b) \rightarrow (A'', T_b'')$*, then there exists* σ_c *such that* $(A', T_a' \parallel T_b) \rightarrow \sigma_c$ *and* $(A'', T_a \parallel T_b'') \rightarrow \sigma_c$.

Proof. We proceed by induction on the derivation of $(A, T_a) \rightarrow (A', T_a')$. We have twelve cases depending on the last rule used to derive $(A, T_a) \rightarrow (A', T_a')$.

- Rule (1). In this case we have $T_a = (skip \parallel T_2)$ and $A' = A$ and $T_a' = T_2$. So we can pick $\sigma_c = (A'', T_2 \parallel T_b'')$ because $(A', T_a' \parallel T_b) = (A, T_2 \parallel T_b)$ and from $(A, T_b) \rightarrow (A'', T_b'')$ and Rule (6) we have $(A, T_2 \parallel T_b) \rightarrow (A'', T_2 \parallel T_b'')$, and because $(A'', T_a \parallel T_b'') = (A'', (skip \parallel T_2) \parallel T_b'')$ and from Rule (1) we have $(A'', (skip \parallel T_2)) \rightarrow (A'', T_2)$, and finally from $(A'', (skip \parallel T_2)) \rightarrow (A'', T_2)$ and Rule (5) we have $(A'', (skip \parallel T_2) \parallel T_b'')) \rightarrow (A'', T_2 \parallel T_b'')$.
- Rule (2). This case is similar to the previous case; we omit the details.
- Rules (3)–(4). Both cases are impossible.
- Rule (5). In this case we have $T_a = T_1 \parallel T_2$ and $T_a' = T_1' \parallel T_2$ and $(A, T_1) \rightarrow (A', T_1')$. From $(A, T_1) \rightarrow (A', T_1')$ and $(A, T_b) \rightarrow (A'', T_b'')$ and the induction hypothesis, we have σ_c' such that $(A', T_1' \parallel T_b) \rightarrow \sigma_c'$ and $(A'', T_1 \parallel T_b'') \rightarrow \sigma_c'$. Let us show that we can pick σ_c such that $(A', (T_1' \parallel T_b) \parallel T_2) \rightarrow \sigma_c$ and $(A'', (T_1 \parallel T_b'') \parallel T_2) \rightarrow \sigma_c$. We have two cases:
  - If $\sigma_c' = \mathsf{error}$, then we use Rule (3) to pick $\sigma_c = \mathsf{error}$.
  - If $\sigma_c' = (A_c, T_c)$, then then we use Rule (5) to pick $(A_c, T_c \parallel T_2)$.

  From $(A', (T_1' \parallel T_b) \parallel T_2) \rightarrow \sigma_c$ and $(A'', (T_1 \parallel T_b'') \parallel T_2) \rightarrow \sigma_c$, the result then follows from $\parallel$ being associative and commutative.
- Rules (6)–(8). All three cases are similar to the case of Rule (1); we omit the details.
- Rule (9). In this case we have $T_a = (item.put(e_1, e_2); s)$ and $A' = A[[\![e_1]\!] := [\![e_2]\!]]$ and $T_a' = s$ and $(A[[\![e_1]\!]] = \bot)$. Let us do a case analysis of the last rule used to derive $(A, T_b) \rightarrow (A'', T_b'')$.
  - Rule (1). In this case we have $T_b = skip \parallel T_2$ and $A'' = A$ and $T_b'' = T_2$. So we can pick $\sigma_c = (A', s \parallel T_2)$ because $(A', T_a' \parallel T_b) = (A', s \parallel (skip \parallel T_2))$ and from Rule (6) and Rule (1) we have $(A', s \parallel (skip \parallel T_2)) \rightarrow (A', s \parallel T_2) = \sigma_c$, and because $(A'', T_a \parallel T_b'') = (A, T_a \parallel T_2)$ and from Rule (5) we have $(A, T_a \parallel T_2) \rightarrow (A', s \parallel T_2) = \sigma_c$.
  - Rule (2). This case is similar to the previous case; we omit the details.
  - Rule (3)–(4). Both cases are impossible.
  - Rule (5). In this case we have $T_b = T_1 \parallel T_2$ and $T_b'' = T_1'' \parallel T_2$ and $(A, T_1) \rightarrow (A'', T_1'')$. We have two cases:
    - If $[\![e_1]\!] \in dom(A'')$, then we can pick $\sigma_c = \mathsf{error}$ because from $(A''[[\![e_1]\!]] \neq \bot)$ and Rule (10) and Rule (5) we have $(A'', T_a \parallel T_b'') \rightarrow \sigma_c$, and because from $(A, T_b) \rightarrow (A'', T_b'')$ and $(A[[\![e_1]\!]] = \bot)$ and $(A''[[\![e_1]\!]] \neq \bot)$ and $(A'[[\![e_1]\!]] \neq \bot)$ and Lemma 5, we have $(A', T_b) \rightarrow \mathsf{error}$, and so from Rule (4) we have $(A', T_a' \parallel T_b) \rightarrow \sigma_c$.

- If $[\![e_1]\!] \notin dom(A'')$, then we define $A_c = A''[[\![e_1]\!] := [\![e_2]\!]]$ and we pick $\sigma_c = (A_c, T'_a \parallel (T''_1 \parallel T_2))$. From $(A[[\![e_1]\!]] = \bot)$ and $(A, T_b) \rightarrow (A'', T''_b)$ and $[\![e_1]\!] \notin dom(A'')$ and Lemma 6, we have $(A', T_b) \rightarrow (A_c, T''_b)$, and then from Rule (6) we have $(A', T'_a \parallel T_b) \rightarrow (A_c, T'_a \parallel T''_b) = \sigma_c$. From Rule (6) and Rule (9) and $[\![e_1]\!] \notin dom(A'')$, we have $(A'', T_a \parallel T''_b) \rightarrow \sigma_c$.
  - Rule (6). This case is similar to the previous case; we omit the details.
  - Rule (7)–(8). Both cases are similar to the case of Rule (1); we omit the details.
  - Rule (9). In this case we have $T_b = (item.put(e'_1, e'_2); s')$ and $A'' = A[[\![e'_1]\!] := [\![e'_2]\!]]$ and $T''_b = s'$ and $(A[[\![e'_1]\!]] = \bot)$. We have two cases:
    - If $[\![e_1]\!] = [\![e'_1]\!]$, then we can pick $\sigma_c = \mathsf{error}$ because $(A'[[\![e_1]\!]] \neq \bot)$ and $(A''[[\![e_1]\!]] \neq \bot)$ and from Rule (10) we have both $(A', T'_a \parallel T_b) \rightarrow \sigma_c$ and $(A'', T_a \parallel T''_b) \rightarrow \sigma_c$.
    - If $[\![e_1]\!] \neq [\![e'_1]\!]$, then we define $A_c = A[[\![e_1]\!] := [\![e_2]\!]][[\![e'_1]\!] := [\![e'_2]\!]]$ and we pick $\sigma_c = (A_c, s \parallel s')$. From $(A[[\![e'_1]\!]] = \bot)$ and $[\![e_1]\!] \neq [\![e'_1]\!]$, we have $(A'[[\![e'_1]\!]] = \bot)$. From Rule (6) and Rule (9) and $(A'[[\![e'_1]\!]] = \bot)$ we have $(A', T'_a \parallel T_b) \rightarrow \sigma_c$. From $(A[[\![e_1]\!]] = \bot)$ and $[\![e_1]\!] \neq [\![e'_1]\!]$, we have $(A''[[\![e_1]\!]] = \bot)$. From Rule (5) and Rule (9) and $(A''[[\![e_1]\!]] = \bot)$ we have $(A'', T_a \parallel T''_b) \rightarrow \sigma_c$.
  - Rule (10). This case is impossible.
  - Rule (11)–(12). Both cases are similar to the case of Rule (1); we omit the details.
- Rule (10). This case is impossible.
- Rules (11)–(12). Both of cases are similar to the case of Rule (1); we omit the details.

□

Lemma 8. (Local Confluence) *If $\sigma \rightarrow \sigma'$ and $\sigma \rightarrow \sigma''$, then there exists σ_c such that $\sigma' \rightarrow^* \sigma_c$ and $\sigma'' \rightarrow^* \sigma_c$.*

Proof. We proceed by induction on the derivation of $\sigma \rightarrow \sigma'$. We have twelve cases depending on the last rule use to derive $\sigma \rightarrow \sigma'$.

- Rule (1). We have $\sigma = (A, skip \parallel T_2)$ and $\sigma' = (A, T_2)$. Let us do a case analysis of the last rule used to derive $\sigma \rightarrow \sigma''$.
  - Rule (1). In this case, $\sigma' = \sigma''$, and we can then pick $\sigma_c = \sigma'$.
  - Rule (2). In this case we must have $T_2 = skip$ and $\sigma'' = (A, skip)$. So $\sigma' = \sigma''$ and we can pick $(A_c, T_c) = (A, skip)$.
  - Rule (3). This case is impossible because it requires $(A, skip)$ to take a step.
  - Rule (4). In this case we have $\sigma'' = \mathsf{error}$ and $(A, T_2) \rightarrow \mathsf{error}$. So we can pick $\sigma_c = \mathsf{error}$ because $(A, T_2) \rightarrow \mathsf{error}$ is the same as $\sigma' \rightarrow \sigma_c$ and $\sigma'' = \sigma_c$.
  - Rule (5). This case is impossible because it requires $skip$ to take a step.
  - Rule (6). In this case we have $\sigma'' = (A', skip \parallel T'_2)$ and $(A, T_2) \rightarrow (A', T'_2)$. So we can pick $\sigma_c = (A', T'_2)$ because from Rule (1) we have $\sigma'' \rightarrow \sigma_c$, and we also have that $(A, T_2) \rightarrow (A', T'_2)$ is the same as $\sigma' \rightarrow \sigma_c$.
  - Rules (7)–(12). Each of these is impossible because $T = skip \parallel T_2$.
- Rule (2). This case is similar to the previous case; we omit the details.
- Rule (3). We have $\sigma = (A, T_1 \parallel T_2)$ and $\sigma' = \mathsf{error}$ and $(A, T_1) \rightarrow \mathsf{error}$. Let us do a case analysis of the last rule used to derive $\sigma \rightarrow \sigma''$.
  - Rule (1). This case impossible because it requires $T_1 = skip$ which contradicts $(A, T_1) \rightarrow \mathsf{error}$.
  - Rule (2). In this case we have $T_2 = skip$ and $\sigma'' = (A, T_1)$. So we can pick $\sigma_c = \mathsf{error}$ because $\sigma' = \sigma_c$ and $(A, T_1) \rightarrow \mathsf{error}$ is the same as $\sigma'' \rightarrow \sigma_c$.
  - Rule (3). In this case, $\sigma' = \sigma''$, and we can then pick $\sigma_c = \sigma'$.
  - Rule (4). In this case, $\sigma' = \sigma''$, and we can then pick $\sigma_c = \sigma'$.
  - Rule (5). In this case we have $(A, T_1) \rightarrow (A', T'_1)$ and $\sigma'' = (A', T'_1 \parallel T_2)$. From the induction hypothesis we have that there exists σ'_c such that $\mathsf{error} \rightarrow^* \sigma'_c$ and $(A', T'_1) \rightarrow^* \sigma'_c$. Given that error has no outgoing transitions, we must have $\sigma'_c = \mathsf{error}$. So we can pick $\sigma_c = \mathsf{error}$ because $\sigma' = \sigma_c$ and because from Lemma 1.1 and $(A', T'_1) \rightarrow^* \sigma'_c$, we have $\sigma'' \rightarrow^* \mathsf{error}$.
  - Rule (6). In this case we have $(A, T_2) \rightarrow (A', T'_2)$ and $\sigma'' = (A', T_1 \parallel T'_2)$. In this case we can pick $\sigma_c = \mathsf{error}$ because $\sigma' = \sigma_c$ and because from $(A, T_1) \rightarrow \mathsf{error}$ and $(A, T_2) \rightarrow (A', T'_2)$ and Lemma 4 and Lemma 3 we have $(A', T_1) \rightarrow \mathsf{error}$, so from Rule (3) we have we have $\sigma'' \rightarrow \sigma_c$.
  - Rules (7)–(12). Each of these is impossible because $T = skip \parallel T_2$.
- Rule (4). This case is similar to the previous case; we omit the details.
- Rule (5). We have $\sigma = (A, T_1 \parallel T_2)$ and $\sigma' = (A', T'_1 \parallel T_2)$ and $(A, T_1) \rightarrow (A', T'_1)$. Let us do a case analysis of the last rule used to derive $\sigma \rightarrow \sigma''$.
  - Rule (1). This case is impossible because for Rule (1) to apply we must have $T_1 = skip$, but for Rule (5) to apply we must have that (A, T_1) can take a step, a contradiction.
  - Rule (2). In this case, we must have $T_2 = skip$ and $\sigma'' = (A, T_1)$. So we can pick $\sigma_c = (A', T'_1)$ because from Rule (2) we have $\sigma' \rightarrow \sigma_c$, and we also have that $(A, T_1) \rightarrow (A', T'_1)$ is the same as $\sigma'' \rightarrow \sigma_c$.
  - Rule (3). In this case we have $(A, T_1) \rightarrow \mathsf{error}$ and $\sigma'' = \mathsf{error}$. From the induction hypothesis we have that there exists σ'_c such that $\mathsf{error} \rightarrow^* \sigma'_c$ and $(A', T'_1) \rightarrow^* \sigma'_c$. Given that error has no outgoing transitions, we must have $\sigma'_c = \mathsf{error}$. So we can pick $\sigma_c = \mathsf{error}$ because $\sigma'' = \sigma_c$ and because from Lemma 1.1 and $(A', T'_1) \rightarrow^* \sigma'_c$, we have $\sigma' \rightarrow^* \mathsf{error}$.
  - Rule (4). In this case we have $(A, T_2) \rightarrow \mathsf{error}$ and $\sigma'' = \mathsf{error}$. So we can pick $\sigma_c = \mathsf{error}$ because $\sigma'' = \mathsf{error}$ and because from $(A, T_2) \rightarrow \mathsf{error}$ and $(A, T_1) \rightarrow (A', T'_1)$ and Lemma 4 and Lemma 3 we have $(A', T_2) \rightarrow \mathsf{error}$, so from Rule (4) we have $\sigma' \rightarrow \mathsf{error}$.
  - Rule (5). In this case we must have $\sigma'' = (A'', T''_1 \parallel T_2)$ and $(A, T_1) \rightarrow (A'', T''_1)$. From $(A, T_1) \rightarrow (A', T'_1)$ and $(A, T_1) \rightarrow (A'', T''_1)$, and the induction hypothesis, we have that there exists σ'_c such that $(A', T'_1) \rightarrow^* \sigma'_c$ and $(A'', T''_1) \rightarrow^* \sigma'_c$. We have two cases.
    1. If $\sigma'_c = \mathsf{error}$, then we can pick $\sigma_c = \mathsf{error}$ because from $(A', T'_1) \rightarrow^* \sigma'_c$ and Lemma 1.1 we have $\sigma' \rightarrow \sigma_c$, and because from $(A'', T''_1) \rightarrow^* \sigma'_c$ and Lemma 1.1 we have $\sigma'' \rightarrow \sigma_c$.
    2. If $\sigma'_c = (A_c, T_c)$, then we can pick $\sigma_c = (A_c, T_c \parallel T_2)$ because from $(A', T'_1) \rightarrow^* \sigma'_c$ and Lemma 2.1 we have

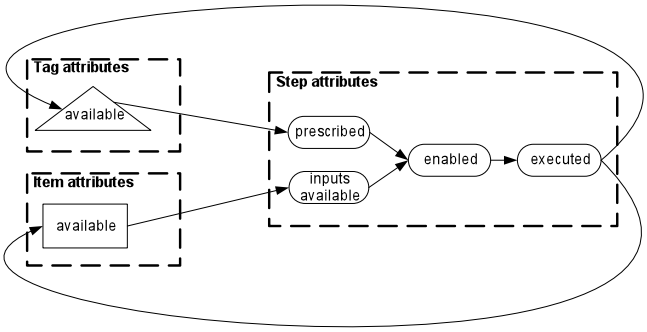

Figure 7. Control, data, and step Attributes

$\sigma' \rightarrow^* \sigma_c$, and because from $(A'', T_1'') \rightarrow^* \sigma_c'$ and Lemma 2.1 we have $\sigma'' \rightarrow^* \sigma_c$.

- Rule (6). In this case we must have $\sigma'' = (A'', T_1 \parallel T_2')$ and $(A, T_2) \rightarrow (A'', T_2')$. From $(A, T_1) \rightarrow (A', T_1')$ and $(A, T_2) \rightarrow (A'', T_2')$ and Lemma 7, we have that there exists σ_c such that $(A', T_1' \parallel T_2) \rightarrow \sigma_c$ and $(A'', T_1 \parallel T_2') \rightarrow \sigma_c$, that is, $\sigma' \rightarrow \sigma_c$ and $\sigma'' \rightarrow \sigma_c$.
- Rules (7)–(12). Each of these is impossible because $T = T_1 \parallel T_2$.

- Rule (6). This case is similar to the previous case; we omit the details.
- Rules (7)–(12). In each of these cases, only one step from σ is possible so $\sigma' = \sigma''$ and we can then pick $\sigma_c = \sigma'$.

□

Lemma 9. (Confluence) *If $\sigma \rightarrow^* \sigma'$ and $\sigma \rightarrow^* \sigma''$, then there exists σ_c such that $\sigma' \rightarrow^* \sigma_c$ and $\sigma'' \rightarrow^* \sigma_c$.*

Proof. We have Lemma 8 and that $\rightarrow$ is strongly normalizing, so the result follows from Newman's Lemma. □

Theorem 1. (Determinism) *If $\sigma \rightarrow^* \sigma'$ and $\sigma \rightarrow^* \sigma''$, and σ', σ'' are both final states, then $\sigma' = \sigma''$.*

Proof. We have from Lemma 9 that there exists σ_c such that $\sigma' \rightarrow^* \sigma_c$ and $\sigma'' \rightarrow^* \sigma_c$. Given that neither σ' or σ'' have any outgoing transitions, we must have $\sigma' = \sigma_c$ and $\sigma'' = \sigma_c$, hence $\sigma' = \sigma''$. □

B. Runtime Semantics and Implementations

B.1 Runtime Semantics

As the program executes, instances of control collections, data collections, and steps monotonically accumulate attributes indicating their state. The set of attributes for these instances and the partial ordering in which an instance can acquire these attributes is shown in Figure 7.

We will not discuss garbage collection here except to say that a data instance or control instance that has been determined to be garbage is attributed as *dead*. It is a requirement of any garbage collection algorithm that the semantics remain unchanged if dead objects are removed.

The *execution frontier* is the set of instances that are of any interest at some particular time, i.e., the set of instances that have any attribute but are not yet *dead* (for data instances and control instances) or*executed* (for steps). The execution frontier evolves during execution.

Program termination occurs when no step is currently executing and no unexecuted step is currently *enabled*. *Valid program termination* occurs when a program terminates and all prescribed steps have executed. The program is deterministic and produces the same results regardless of the schedule within or the distribution among processors. It is possible to write an invalid program, one that stops with steps that are prescribed but whose input is not available. However, altering the schedule or distribution will not change this result. Note that the semantics allow but do not imply parallel execution. This makes it easier to develop and debug an application on a uniprocessor.

The Concurrent Collections execution model can be seen as a natural convergence of the data flow and program dependence graph execution models. The producer-consumer relations established via data collections support a general data flow execution model, whereas control collections and prescriptions can be used to enforce control dependences and correspond to region nodes in program dependence graphs.

B.2 Runtime System Design Space

The CnC semantics outlined in the previous section allow for a wide variety of runtime systems. They vary along several dimensions: The target architecture determines the types of parallelism supported and whether the target has shared or distributed memory. The 3 basic decisions when mapping a given program to a given architecture include: choice of grain, distribution among computational resources, scheduling within a computational resource. The runtimes vary depending on whether these decisions are made statically or dynamically. In addition, a runtime might include additional capabilities such as checkpoint/restart, speculative execution, demand-driven execution, auto-tuning, etc.

We have explored the following points in the CnC design space:

- distributed memory, static grain, static distribution among address spaces, dynamic schedule within an address space, on MPI
- distributed memory, static grain, static distribution among address spaces, dynamic schedule within an address space, on MPI, with a checkpoint/restart capability
- distributed memory, static grain, static distribution among address spaces, static schedule within and address space, on MPI
- shared memory, dynamic grain, dynamic distribution among cores, dynamic schedule within cores, on MPI
- shared memory, static grain, dynamic distribution among cores, dynamic schedule within cores, in X10
- shared memory, static grain, dynamic distribution among cores, dynamic schedule within cores, on TBB

Of the runtimes listed above, the last two represent recent work for targeting multicore SMPs and are the focus of this paper.

The runtimes above operate on distinct representations. For each representation, a translator can be used to convert the CnC textual form to that representation. We have built a translator for the C/C++ runtime that converts to the appropriate classes described below. In addition, for each step collection, it creates a hint, i.e., a template of the step based on information in the graph specification. The hint for a step specifies the form for the step, as well as the form for its gets and puts. The user fills in the local step computation. For now, the HJ forms are hand-generated but a similar translator is planned.

B.3 Runtime based on C++ and Threading Building Blocks

The Concurrent Collection model allows for a wide variety of runtime systems as described above. The implementations discussed in this paper target shared memory multi-core processors, and are

characterized by static grain choice, dynamic schedule and dynamic distribution. One approach consists of a small C++ class library built on the Intel® Threading Building Blocks (TBB) [8]. TBB controls the scheduling and execution of the program. There are several advantages in implementing Concurrent Collections on TBB. TBB supports fine-grain grain parallelism with tasks. TBB tasks are user-level function objects which are scheduled by a work-stealing scheduler. A work-stealing scheduler provides provably good performance for well-structured programs; the TBB scheduler is inspired by Cilk [2]. TBB also supports a set of concurrent containers, including vectors and hash-maps. These provide functionality similar to the containers in the Standard Template Library [26], and they also permit fast concurrent access. Finally, TBB provides a scalable concurrent memory allocator [17], which addresses a common bottleneck in parallel systems.

Classes represent the Concurrent Collection objects. A Graph class represents the program; it contains objects that represent the steps, data items, and control instances and their relationships. A step is represented as a user-written C++ function wrapped in a function object. When a control instance that prescribes a step instance is created, an instance of the step functor is created and mapped to a TBB task. Get and Put APIs enable the user function to read and write data instances. The runtime provides classes to represent the three types of collections in a Concurrent Collections program (TagCollection, StepCollection, and DataCollection). A ControlCollection maintains a list of all the StepCollections that it prescribes. When a ControlCollection Put method is called with a tag, a StepInstance is created for each the prescribed StepCollections. These StepInstances are mapped into TBB tasks and scheduled. Data items are created by calling an DataCollection Put method with a value and an associated tag. Items are retrieved by calling an DataCollection Get method with a data tag. The data items for each DataCollection are maintained in a TBB hash-map, accessed by tag.

We schedule StepInstances speculatively, as soon as they are prescribed by a tag. When a StepInstance calls an DataCollection Get method with a data tag, the value associated with the tag may have not yet been created, i.e., it may not have the attribute *inputs-available* and therefore may not yet have the attribute *enabled*. If this is the case, we queue the StepInstance on a local queue associated with the tag, and release it to the TBB scheduler when the value associated with the tag is Put by another step. When we re-schedule the StepInstance, we restart it from the beginning; here we exploit the functional nature of Concurrent Collection steps. Notice that each StepInstance attempts to run at most once per input item.

B.4 Runtime based on Habanero-Java

In this section, we discuss our experiences with an implementation of the CnC programming model in the Java-based Habanero-Java (HJ)[2] programming language being developed in the Habanero Multicore Software Research project at Rice University [16] which aims to facilitate multicore software enablement through language extensions, compiler technologies, runtime management, concurrency libraries and tools. The HJ language is an extension to v0.41 of the X10 language described in [6]. The X10 language was developed at IBM as part of the PERCS project in DARPA's High Productivity Computing Systems program. The initial versions of the X10 language (up to v1.5) used Java as its underlying sequential language, but future versions of X10 starting with v1.7 will move to a Scala-based syntax [29] which is quite different from Java.

[2] Habanero-C, a C++ based variant of this language, is also in development.

Some key differences between HJ and v0.41 of X10 are as follows:

- HJ includes the *phasers* construct [25], which generalizes X10's clocks to unify point-to-point and collective synchronization.
- HJ includes a new *delayed async* construct, described below in more detail.
- HJ permits dynamic allocation of *places*.
- HJ extends X10's *atomic* construct so as to enable enforcement of mutual exclusion across multiple places.

Since HJ is based on Java, the use of certain primitives from the Java Concurrency Utilities [22] is also permitted in HJ programs, most notably operations on Java Concurrent Collections such as `java.util.concurrent.ConcurrentHashMap` and on Java Atomic Variables.

One of the attractive properties of the CnC model is that it has a very small number of primitives. We were pleasantly surprised to see how straightforward it has been to map CnC primitives to HJ, as summarized in Table 3. The following subsections provide additional details for this mapping.

B.4.1 Tags

We used Java $String$ and X10 $point$ constructs to implement Tags. A $point$ in X10 is an integer tuple, that can be declared with an unspecified rank. A multidimensional tag is implemented by a multidimensional point.

B.4.2 Prescriptions

When a step needs to put a prescription tag in the control collection, we create a custom closure object for each of the steps prescribed by that tag, which we then pass to the runtime to store for immediate or later execution, depending on the scheduling policy. Using the blocking gets policy, the step is spawned immediately using a normal $async$ and it will block when trying to get a data item that is not yet available. Using the roll-back and replay policy, the step is spawned immediately using a normal *async*, when it attempts to get an item that is not yet available it will throw an exception, causing the execution to abort, and the closure object of the step to be put on a queue waiting for the data item to appear. When a corresponding put is performed, the steps in the queue are restarted. Using a non-blocking gets policy, the step is spawned using a *delayed async*. The normal async statement, *async* ⟨*stmt*⟩, is derived from X10 and causes the parent activity to create a new child activity to execute ⟨*stmt*⟩. Execution of the async statement returns immediately i.e., the parent activity can proceed immediately to its next statement.

The delayed async statement, *async (*⟨*cond*⟩*)* ⟨*stmt*⟩, is similar to a normal async except that execution of ⟨*stmt*⟩ is guaranteed to be delayed until after the boolean condition, ⟨*cond*⟩, evaluates to true.

B.4.3 Data Collections

We use the `java.util.concurrent.ConcurrentHashMap` class to implement data collections. Our HJ implementation of data collections supports the following operations:

- `new DataCollection(String` *name*`)`: create and return a new data collection. The string parameter, *name*, is used only for diagnostic purposes.
- C.`put(point` p`, Object` O`)`: insert data item O with tag p into collection C. Throw an exception if C already contains an item with tag p.

CnC construct	Translation to HJ
Tag	Java *String* object or X10 *point* object
Prescription	*async* or *delayed async*
Item Collection	`java.util.concurrent.ConcurrentHashMap`
put() on Item Collection	Nonblocking *put()* on `ConcurrentHashMap`
get() on Item Collection	Blocking or nonblocking *get()* on `ConcurrentHashMap`

Table 3. Summary of mapping from CnC primitives to HJ primitives

- C.`awaitAndGet(point` p`)`: return data item in collection C with tag p. If necessary, the caller blocks until item becomes available.
- C.`containsTag(point` p`)`: return true if collection C contains a data item with tag p, false otherwise.
- C.`get(point` p`)`: return data item in collection C with tag p if present; return `null` otherwise. The HJ implementation of CnC ensures that this operation is only performed when tag p is present *i.e.,* when C.`containsTag(point` p`) = true`. Unlike `awaitAndGet()`, a `get()` operation is guaranteed to always be nonblocking.

B.4.4 Put and Get Operations

A CnC `put` operation is directly translated to a `put` operation on an HJ data collection, but implementing `get` operations can be more complicated. A naive approach is to translate a CnC `get` operation to an `awaitAndGet` operation on an HJ data collection. However, this approach does not scale well when there are a large number of steps blocked on `get` operations, since each blocked activity in the current X10 runtime system gets bound to a separate Java thread. A Java thread has a larger memory footprint than a newly created async operation. Typically, a single heavyweight Java thread executes multiple lightweight async's; however, when an async blocks on an `awaitAndGet` operation it also blocks the Java thread, thereby causing additional Java threads to be allocated in the thread pool[10]. In some scenarios, this can result in thousands of Java threads getting created and then immediately blocking on `awaitAndGet` operations.

This observation lead to some interesting compiler optimization opportunities of `get` operations using delayed asyncs. Consider a CnC step S that performs two `get` operations followed by a `put` operation as follows (where T_x, T_y, T_z are distinct tags):

$$S\text{: } \{\ x := C.\texttt{get}(T_x);\ y := C.\texttt{get}(T_y);\ z := F(x,y);\\ C.\texttt{put}(T_z, z);\ \}$$

Instead of implementing a prescription of step S with tag T_S as a normal async like "`async` $S(T_S)$", a compiler can implement it using a delayed async of the form "`async await(`C.`containsTag(`T_x`)` `&&` C.`containsTag(`T_y`))` $S(T_S)$". With this boolean condition, we are guaranteed that execution of the step will not begin until data items with tags T_x and T_y are available in collection C.

C. Implementing CnC on top of Java and .NET

Please see Figure 8 for an overview of the translation process of the CnC program into HJ interfaces and classes.

C.1 Parser

The parser parses the grammar shown in Figure 9 and produces an AST as described in Section C.2.

C.2 AST

The AST definition is given in Figure 10.

C.3 Semantic Analysis

The translator checks that the declarations match the usage and global graph conditions such as:

Error The step collection is not prescribed.

Error The step collection is prescribed by more than one control collection.

Error The control collection prescribes a step collection but is not produced.

Error The control collection is consumed by the environment but is not produced.

Error The data collection is consumed but not produced.

Warning The control collection is produced but does not prescribe a step collection and is not consumed by the environment.

Warning The control collection is produced by the environment and consumed by the environment but is not otherwise referenced.

Warning The data collection is produced but not consumed.

Warning The data collection is produced by the environment and consumed by the environment but is not otherwise referenced

C.4 CodeGen

In this section we describe how to generate code for a HJ CnC application. We use the following HJ CnC program as a running example:

```
<Control1: int a, int b> :: (Step1: int a, int b);
<Control2: int x> :: (Step2);
<Control2: int y> :: (Step3);

[A: int] -> (Step1) -> [B: int], <Control 2: int x>
[B] -> (Step2)
(Step3) -> [C]
```

The code generator generates a class for the HJ CnC graph, a class for each control collection used in the program, and a class for each step defined by the user.

C.4.1 Steps

For every step specified in the input program, we generate an abstract class that holds the computation for that step. This abstract class derives from the `Step` abstract class described in Section C.7. For a step named `Step1`, we generate an abstract class `AStep1` that derives from the `Step` class and adds an additional abstract`compute` method. The signature of the compute method depends on the inputs and outputs of the step. The collections read and written by the step get passed as parameters to the `compute` method as `InputCollection`, `OutputCollection`, and `ControlCollection` as appropriate. If a collection is both read and written by the step then it is be passed as an `InputCollection` and as an `OutputCollection` to the step. In addition, the first parameter is a `String` or `Point` object specifying the control instance for the step being executed. An example is shown below.

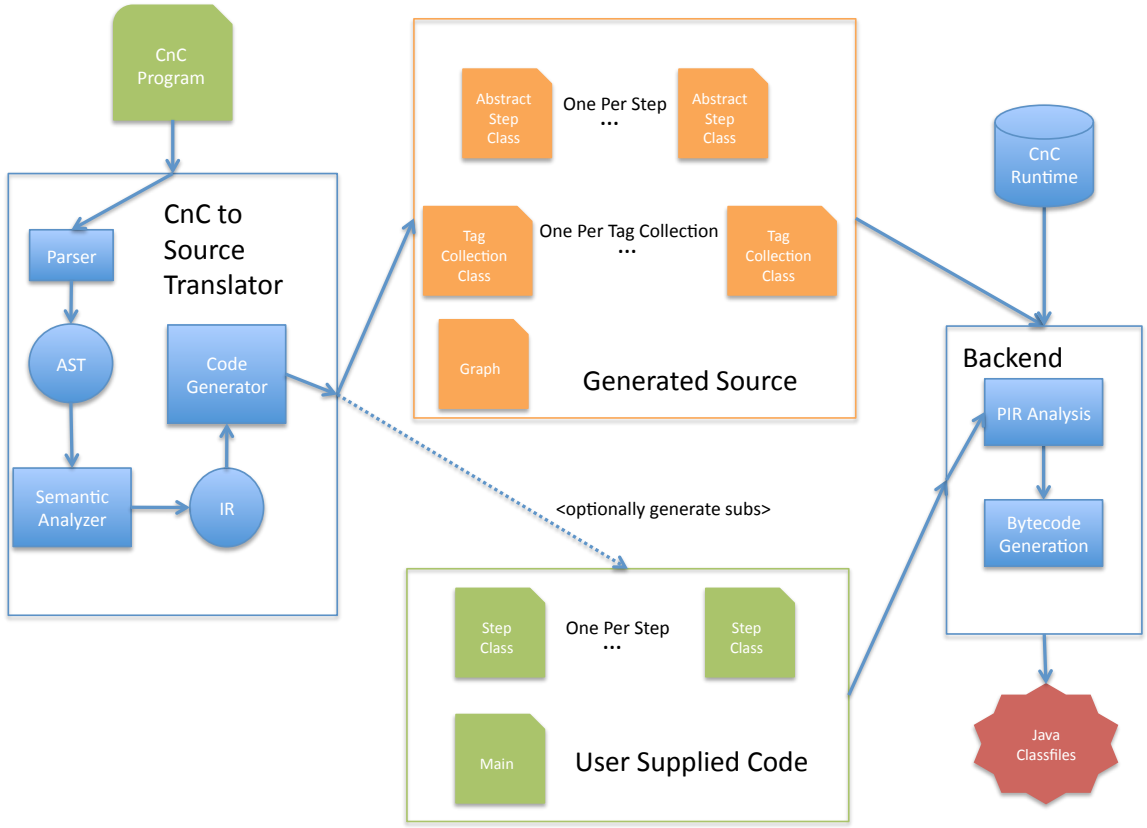

Figure 8. Overview of the translation process of the CnC program into HJ interfaces and classes

```
abstract class AStep1 < Step {
    abstract CnCRet compute(Point Control\_tag, InputCollection A,
                                    OutputCollection B,
                                    ControlCollection Control2)
}
abstract class AStep2 < Step {
    abstract CnCRet compute(Point Control\_tag, InputCollection B)
}
abstract class AStep3 < Step {
    abstract CnCRet compute(Point Control\_tag, OutputCollection C)
}
```

C.4.2 Control Collections

For each control collection referenced in the program, we generate a class that implements the `ControlCollection` interface. The generated class has an instance variable for the graph and one variable for each step that the control collection prescribes. It has a constructor that takes the graph and all the prescribed steps as arguments.

We also generate an implementation of the `Put` method for the class. The put method takes a control instance as its single parameter and generates a call into the runtime passing a `Closure` for the work that should be done when the step executes. The closure is passed as an anonymous inner class that implements two methods: `ready` and `compute`. These methods are implemented by simply forwarding the calls to the step with the appropriate arguments. An example of the generated code is shown below.

```
class Control1Collection implements ControlCollection {
    Graph  g;
    AStep1 step1;
    Tag1(Graph g_, AStep1 step1_) {
    g = g_; step1 = step1_;
    }
    Put(Point tag) {
        CnCRuntime.PutTag(
            new Closure () {
                bool   ready()  {step1.ready(g1, tag)}
                CnCRet compute(){
                  step1.compute(tag, g.A, g.B, g.Tag2)
                }
            }
        )
    }
}

class Control2Collection implements ControlCollection {
    Graph  g;
    AStep2 step2;
    AStep3 step3;

    Tag2(Graph g_, AStep2 step2_, AStep3 step3_) {
        g = g_; step2 = step2_; step3 = step3_;
    }

    Put(Point tag) {
        CnCRuntime. PutTag(
            new Closure () {
                bool   ready()  {step2.ready(g1, tag)}
                CnCRet compute(){step2.compute(tag, g.B)}
            }
        )
        CnCRuntime. PutTag(
```

```
            new Closure() {
                bool ready()     {step3.ready(g1,tag);}
                CnCRet compute() {step3.compute(tag, g.C);}
            }
        )
    }
}
```

C.4.3 Graph

We generate one class for the entire graph. This class has fields for each control collection and data collection in the graph. The graph has one factory method that takes an argument for each step defined by the user.

```
class Graph {
    ControlCollection Tag1;
    ControlCollection Tag2;
    DataCollection A,B,C;

    Graph(AStep1 step1, AStep2 step2, AStep3 step3) {
        Tag1 = new Control1Collection(self, step1)
        Tag2 = new Control2Collection(self, step2, step3);
        DataCollection A,B,C = ...;
    }

}
```

C.5 Program Analysis

After the user has supplied the details of the steps, all of the generated source code and user code can be compiled together. At this time the compiler can perform the CnC program analysis to fill in the details of the `ready` method on the user step classes. (Alternatively, the user could supply an implementation of this method and we can skip the compiler analysis). The compiler analysis should read the body of the user supplied `compute` method and look for all of the `Get` messages sent to data collections. It can use this analysis to generate a predicate for when the step is ready to execute and implement the `ready` method to return an evaluation of that predicate.

C.6 User Responsibilities

The user must supply the implementation of the `compute` method for each step. The user may optionally supply an implementation of the `ready` method for a step to improve system performance using the delayed async construct.

C.7 HJ CnC API

C.8 User Visible API

```
interface ControlCollection {
    Put(Point tag);
}
interface InputCollection {
    Object Get(Point tag);
}
interface OutputCollection {
    void Put(Point tag, Object val);
}
interface DataCollection
  implements InputCollection, OutputCollection {
}
abstract class Step {
    // general methods for gathering statistics or what not
    // body of ready() to be replaced with PIR compiler analysis
    bool   ready(Graph g, Point tag) {return true};
}
```

C.9 Code-Generator Visible API

```
interface CnCRuntime {
    PutTag(Point tag,  Closure computation);
}
interface Closure() {
    bool ready();
    bool compute();
}
```

Grammar

See Figure 9.

AST

Figure 10 shows the definitions of the AST nodes for the .NET CnC implementation. The definitions are written in F#.

```
graph: statements

statements:
    terminated_declaration
    terminated_relation
    statements terminated_declaration
    statements terminated_relation

terminated_declaration:
    declaration ;

declaration:
    data_declaration
    control_declaration
    step_declaration

terminated_relation:
    relation ;

relation:
    step_execution
    step_prescription

step_execution:
    instance_list -> step_instance
    instance_list -> step_instance -> instance_list
    instance_list <- step_instance
    instance_list <- step_instance <- instance_list
    step_instance <- instance_list
    step_instance -> instance_list

step_prescription:
    control_instance :: step_instance_list
    step_instance_list :: control_instance

instance_list:
    data_instance
    control_instance
    instance_list , data_instance
    instance_list , control_instance

control_instance:
    < T_NAME control_description >
    < T_NAME >

control_description:
    : control_component_list

control_component_list:
    control_component
    control_component_list , control_component

control_component:
    T_NAME T_NAME
    T_TYPE T_NAME
    T_NAME
    T_NAME T_NAME ( param_list )
    T_TYPE T_NAME ( param_list )
    T_NAME ( param_list )

param_list:
    param
    param_list , param

param:
    T_NAME

data_instance:
    [ data_definition control_description ]
    [ data_definition ]

data_definition:
    T_NAME T_NAME
    T_TYPE T_NAME
    T_NAME

step_instance_list:
    step_instance
    step_instance_list , step_instance

step_instance:
    ( T_NAME )
    ( T_NAME control_description )
    T_ENV

data_declaration:
    data_instance attribute_list
    data_instance

control_declaration:
    control_instance attribute_list
    control_instance

step_declaration:
    step_instance attribute_list
    step_instance

attribute_list:
    attribute
    attribute_list , attribute

attribute:
    T_NAME = T_NAME
    T_NAME = T_NUMBER
    T_NAME = T_QUOTEDVAL
```

Figure 9. HJ CnC Grammar

```
type Graph = Statement list

(* Statement *)
and Statement =
    | Declaration of Declaration
    | Relation    of Relation

(* Declaration *)
and Declaration =
    | DataDecl of DataDeclaration
    | controlDecl  of controlDeclaration
    | StepDecl of StepDeclaration

(* Relation *)
and Relation =
    | StepExecution    of StepExecution
    | StepPrescription of StepPrescription

(* Step Execution *)
and StepExecution =
    (Instance list * StepInstance * Instance list)

and Instance =
    | DataInst of DataInstance
    | controlInst  of controlInstance

(* Step Prescription *)
and StepPrescription =
    controlInstance * (StepInstance list)

(* control Instances *)
and controlInstance  = {
    Name : string;
    Tag  : TagDescription;
}
and TagDescription = TagComponent list
and TagComponent =
    | BaseTypeComponent     of BaseTypeTagComponent
    | FunctionTypeComponent of FunctionTypeTagComponent
and BaseTypeTagComponent = {
    ComponentType : Type;
    ComponentName : string;
}
and FunctionTypeTagComponent = {
    FunctionName : string;
    ReturnType : Type;
    ParameterList: string list;
}
```

```
(* Data Instance *)
and DataInstance = {
    Name : string;
    Type : Type;
    Tag : TagDescription;
}

(* Step Instances *)
and StepInstance =
    | UserDefinedStep of UserStepInstance
    | Env
and UserStepInstance = {
    Name : string;
    Tag  : TagDescription;
}

(* Declarations *)
and DataDeclaration = DataInstance
and controlDeclaration  = controlInstance
and StepDeclaration = StepInstance
and Type = string
```

Figure 10. .NET AST