RICE UNIVERSITY

Asynchronous Checkpoint/Restart for the Concurrent Collections Model

by

Nick Vrvilo

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Master of Science

Houston, Texas

August, 2014

ABSTRACT

Asynchronous Checkpoint/Restart
for the Concurrent Collections Model

by

Nick Vrvilo

It has been claimed that what simplifies parallelism can also simplify resilience. In this thesis we describe an asynchronous checkpoint/restart framework for the Concurrent Collections programming model (CnC)—a dataflow-based programming model—and demonstrate that CnC is an exemplar target for a simple yet powerful resilience system for parallel computations. We claim that the same attributes that simplify reasoning about parallel applications written in CnC similarly simplify the implementation of a checkpoint/restart system within the CnC runtime. We define the properties of CnC in the context of a formal executable model built in K. To demonstrate how these simplifying properties of CnC help to simplify resilience, we have implemented a simple checkpoint/restart system within Rice's Habanero-C implementation of the CnC runtime. We show how the CnC runtime can fully encapsulate checkpointing and restarting processes, enabling application programmers to gain all the benefits of resilience without any added effort beyond implementing an application in CnC. Furthermore, our approach is asynchronous and thus avoids synchronization overheads present in traditional techniques.

Acknowledgments

I would like to thank my advisor, Vivek, for his help and guidance in writing this thesis, along with John and Swarat for their contributions as members of my thesis committee. I also thank our collaborators from the Intel CnC team—Kath Knobe and Frank Schlimbach—for all of their input and support in moving this research forward. Finally, I thank my family—especially my wife, Siang-Wei—for all of their love and encouragement in getting to where I am today.

This work was supported in part by the DOE-funded Traleika Glacier X-Stack project (Intel CS1924113), and by the Data Analysis and Visualization Cyberinfrastructure funded by NSF grant OCI-0959097.

Contents

List of Figures

List of Listings

List of Rewrite Rules

Chapter 1

Introduction

This thesis defines the semantics and runtime properties of a checkpoint/restart (C/R)-based resilience system targeting the Habanero Concurrent Collections (CnC) runtime platform. We investigate the claim that *"what is good for parallelization is good for resilience"* [1] and demonstrate that the properties that make CnC well suited for parallelization indeed simplify the implementation of C/R. We present a formal model to demonstrate the correctness of our resilience technique, as well as an implementation of C/R in the Habanero-C CnC runtime (CnC-HC).

1.1 Motivation for CnC C/R

While supercomputers today may lose a node due to a hard error about once a week, exascale supercomputers expected by 2020 will have a mean time between failures on the order of minutes, making checkpointing "both more critical and less practical" on these systems [2]. Restoring a distributed application with a large memory footprint from disk can take several minutes in practice [3]; therefore, it is not far-fetched that the overhead incurred by current C/R techniques would consume nearly 100% of the computation time on future exascale systems. Reducing synchronization without sacrificing correctness is an important goal of all modern resilience systems, and will continue to be as supercomputing clusters continue to grow. Since several aspects of the CnC programming model make it a good fit for extreme scale computing [4], we should also explore the possibilities CnC or a similar programming model brings to the realm of resilience. As the first step in this endeavor, this thesis demonstrates how to design a C/R system for CnC in a shared memory environment.

1.2 Thesis Statement

Since the Concurrent Collections programming model has many properties that simplify the process of expressing parallel programs, it is also an ideal target for a simple and efficient checkpoint/restart-based resilience system.

1.3 Contributions

This thesis makes the following contributions. We define the semantics of a basic CnC runtime in a formal model, and use that model to prove key properties of the CnC runtime. We define the semantics of a CnC checkpoint as it receives updates from a running graph execution, and demonstrate that the checkpoint can be used to correctly restart computation. We define the concept of an execution frontier, and update the runtime and checkpoint models to incorporate concepts from execution frontiers. Finally, we demonstrate an implementation of CnC C/R in the CnC-HC runtime.

1.4 Organization

Chapter 2 provides background on the concepts used throughout the rest of the thesis and gives context with related work. Chapter 3 defines a small API that we use to write sample applications that can run against our executable models of CnC. Chapter 4 introduces our model for the basic CnC runtime, and proves that some key properties of the runtime in the context of that model. Chapter 5 defines the model for a CnC checkpoint, and outlines the method for restarting a CnC execution graph from a valid checkpoint. Chapter 6 introduces the concept of an execution frontier and modifies the runtime model from chapter 4 to incorporate execution frontiers. Chapter 7 updates the checkpoint model from chapter 5 to work with the new runtime model from chapter 6. Chapter 8 explores how to add checkpoint/restart support to the existing Habanero-C implementation of CnC based off the models from chapters 6 and 7. Chapter 9 presents our conclusions and directions for future work.

Chapter 2

Background

This chapter provides background information about the programming models and frameworks discussed in this thesis. In section 2.1 we explain the Concurrent Collections (CnC) programming model, for which we design and implement a checkpoint/restart system in this thesis. In section 2.2 we discuss current approaches to checkpoint/restart, comparing them to our checkpoint/restart strategy for CnC. Section 2.3 gives a brief description of the Habanero-C framework, which is the basis for the CnC implementation used in chapter 8. Finally, section 2.4 describes the K framework; K is a rewrite-based executable semantic framework that we use to model the CnC runtime and checkpoint/restart behavior.

2.1 Concurrent Collections (CnC)

CnC is a system for describing the structure of parallel computation, or coordinating the data- and control-flow between the individual steps of a computation [5, 6]. A CnC application specifies a set of discrete step functions, and the data collections used as input to and output from those step functions.[1] The CnC coordination language describes the relationship between a specific invocation of a step function, its input

[1] This model varies slightly from the traditional CnC model in that it lacks *control collections*; however, this elision in our model reflects the absence of control collections in the Habanero variants of CnC developed at Rice University, on which this work is based. For a brief overview of control collections, and a discussion of the equivalence of this simplified CnC model with the traditional model, please see appendix A.

and output data, as well as parent/child relationships between to other step function invocations. The remainder of this section provides a general overview of the CnC programming model, outlining the key properties of CnC, and how these properties are particularly useful when implementing checkpoint/restart (C/R) for CnC.

2.1.1 Key Properties of CnC

In this thesis, we leverage several distinctive characteristics of the CnC programming model, which give our C/R solution a number of interesting and unique properties. These are the characteristics that make the CnC programming model well suited for expressing large-scale parallel computations. In this section, we outline five key characteristics of CnC: graph representation, single-assignment data, monotonically growing state, discrete computation steps, and side-effect-free computation steps. We leverage these five characteristics for C/R.

Graph Representation of the Application

A fundamental characteristic of a CnC application is that the entire computation flow is represented as a graph. An application is partitioned into collections of computation steps and data items, each of which describe a class of step (computation) or item (data) instances. These collections serve as the nodes of the graph. The *prescribe* (step creation), *put* (item creation) and *get* (item read) relationships among the collections are represented as edges in the graph. Figure 2.1 shows a graph representation for a simple CnC application. By default, we mean a static program graph, when referring to a CnC graph. When necessary, we will differentiate between a static CnC graph, which defines a CnC program, and a dynamic CnC graph, which defines a CnC program execution.

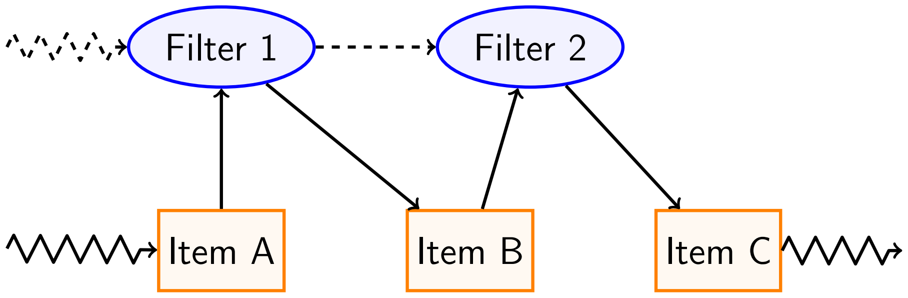


Figure 2.1 : The abstract graph representation of a simple data-filtering CnC application. The two ellipses represent step collections for two separate levels of filtering. The three rectangles represent item collections for holding all of the input data (*Item A*), the results of the first filter pass (*Item B*), and the final output from the second filter pass (*Item C*). Solid edges represent puts to and gets from item collections. Dashed edges represent prescriptions (creation) of new step instances. Jagged edges represent the interactions with the application environment that encloses the CnC graph.

By providing a high-level graphical representation of the application, the user provides the CnC runtime with all the necessary information to automatically track the incremental progress of the application. In the case of a failure, the runtime can restart the computation by simply restarting all of the computation steps that were running at the time of the failure, and providing the input data for those steps to run to completion.

Since all of the information needed to perform a restart is encapsulated within the CnC computation graph, we are also able to limit the data stored in checkpoints to only the data represented by the graph, rather than saving a full memory dump as done in many other solutions.

Single-Assignment Data and Monotonically Growing State

In a traditional imperative computation model, an application calculates incremental solutions to a problem by updating (or mutating) its in-memory data, eventually resulting in the final output. In such a system, it is possible to checkpoint an inconsistent state if some data is updated during the process of saving the checkpoint, such that part of the checkpoint reflects the update and part does not. In contrast, CnC takes a functional rather than imperative approach to modeling state. All data available at the level of the CnC computation graph is single-assignment, meaning that once a data item is created it is never updated. An individual computation step is free to mutate data local to that step, but all such mutations must be fully encapsulated within the step.

A property that follows from the single-assignment property is the monotonicity property. Since data cannot be updated after appearing in the graph, the overall state of a CnC application appears to only add new data, never removing or mutating previous data. A CnC implementation can optionally free data that is no longer required, though this process can, in general, be more complicated than garbage collection in functional languages [7].

This combination of the single-assignment and monotonicity properties allows us to safely create checkpoints asynchronous of all step computations. Once a data item is created it cannot change in the midst of saving a checkpoint and potentially corrupt the consistency of the checkpoint. Additionally, since updates to a checkpoint only add new information, it is impossible to accidentally remove information that may be necessary to successfully restart. However, some implementations of CnC include mechanisms for removing data after its last use [8, 7], which must be adapted carefully in the presence of checkpointing.

Discrete and Side-Effect-Free Computation Steps

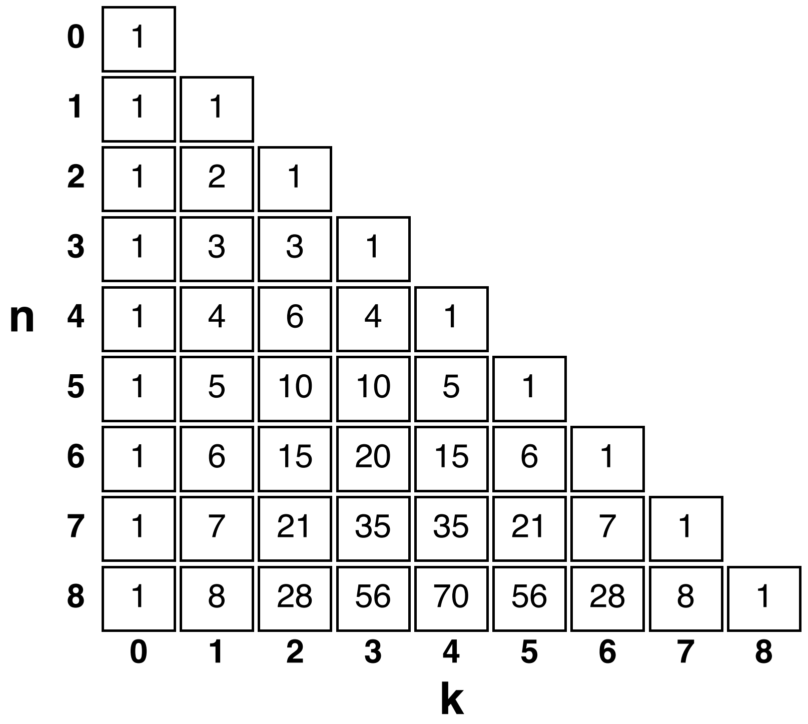
A traditional application may be implicitly divided into several logical computations, but the CnC programming model makes these divisions explicit. The CnC runtime takes advantage of discrete computation steps to run computation steps in parallel on multicore hardware. The fact that CnC applications have discrete computation steps with explicit inputs allows us to restart any given computation at the CnC step granularity. In addition, computation steps in CnC are side-effect-free because the only observable outputs of CnC steps are their items put and steps prescribed. This means that if a computation failed mid-step, there is no possibility that some incremental updates made by the step will corrupt the global CnC graphs state. These properties allow us to safely restart an application that failed at any point in the computation.

Summary

CnC applications have the unique properties of being specified as computation graphs, having single-assignment, a monotonically growing state, and computation steps that are both discrete and side-effect-free. The combination of all these properties gives our C/R solution several unique properties. We can limit the data saved to the information encapsulated within the CnC runtime, checkpointing that data asynchronously and continuously. Finally, we can safely restart a CnC application that failed at any point during the computation.

2.1.2 Pascal's Triangle: A Sample CnC Application

To better describe the CnC programming model, we now introduce a simple CnC application as an example. This sample application computes binomial coefficients—



n \ k	0	1	2	3	4	5	6	7	8
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
6	1	6	15	20	15	6	1		
7	1	7	21	35	35	21	7	1	
8	1	8	28	56	70	56	28	8	1

Figure 2.2 : The first nine rows of Pascal's Triangle. The entries of Pascal's Triangle correspond to the binomial coefficients, such that the entry at row n, column k is equal to ${}_nC_k$.

i.e., the values of ${}_nC_k$ (n choose k)—via Pascal's Triangle.

Review of Pascal's Triangle

Figure 2.2 shows the first nine rows of Pascal's Triangle. If $P(n, k)$ is the value at row n, column k of Pascal's Triangle (where both the row and column numbers are zero-based), then for all $n \geq k \geq 0$:

$$P(n, 0) = P(n, n) = 1 \tag{2.1a}$$

$$P(n, k) = P(n-1, k-1) + P(n-1, k) \tag{2.1b}$$

Equations (2.1a) and (2.1b) exactly match the recursive definition for the binomial coefficients [9]; hence, the entry of Pascal's Triangle at row n, column k corresponds to the binomial coefficient ${}_nC_k$ [10].

Structure of the CnC Graph

As explained earlier in this section, every CnC application must specify a set of step collections, corresponding to the functions used in computation, and a set of item collections, corresponding to the data on which the steps operate. To compute the value of ${}_nC_k$, we must compute n rows and k columns of Pascal's Triangle. Since the only type of data we use in this computation (both for building the triangle and in the result) is the set of values in the triangle, we only need a single item collection to hold that data, which we can call *pascal-entries*. Since we have two different equations for computing the entries of the triangle, we have one step collection for computing values based on equation (2.1a), and another based on equation (2.1b). We call the first step collection *edge-step* since it computes the values along the left and right edges of the triangle, and the second *inner-step* since it computes the remaining values inside the triangle. A high-level sketch of the CnC graph for this application is illustrated in figure 2.3.

In CnC, instances of step and item collections are differentiated by a unique *tag*, often represented by an integer tuple; however, to differentiate step and item collections, we typically refer to the tag of an item instance as a *key*. We identify instances of both the item and step collections by the row and column of the corresponding entry in Pascal's Triangle; therefore, the tags and keys for instances in all three collections are integer pairs of the form $\langle row, col\rangle$. For simplicity, we use the notation $(\!|\mathsf{S}\colon \mathsf{T}|\!)$ to denote an instance of step collection S with the tag T, where the round brackets correspond to the round nodes used for steps in the graphical representation (as shown in figure 2.3). Similarly, we use the notation $[\![\mathsf{I}\colon\mathsf{K}]\!]$ to denote an instance of item collection I with the key K, or $[\![\mathsf{I}\colon\mathsf{K}\rightarrow\mathsf{V}]\!]$ to denote that the item instance has the value V, where the square brackets correspond to the rectangular nodes used for items in the graphical representation.

To give our application a more dynamic feel, each step instance with tag $\langle row, col\rangle$ prescribes the step instance with tag $\langle row{+}1, col\rangle$. Since each row of Pascal's Triangle has one more column than the previous row, steps with tags where $row = col$ also need to prescribe the step with tag $\langle row + 1, col + 1\rangle$. Each step instance also puts a single data item to the *pascal-entries* collection, with a key matching the step's tag, and the value ${}_{row}C_{col}$. Figure 2.4 illustrates these relationships among the step and item collections, with the mapping between step tags and item keys shown explicitly.

It is often useful in a CnC application to parameterize some aspects of the graph structure. For example, one might want to parameterize the dimensions of the input matrices to a matrix kernel in order to make the code more generic. In our application, we want to parameterize the values n and k, which allows us to stop computation at row n of Pascal's triangle. We do this by setting values for n and k in the CnC graph's context, which is available to all CnC functions. These parameters are considered constant throughout the graph execution. The step functions in our application use these parameter values to compute whether or not to prescribe a new step instance corresponding to the next row of the triangle.

Executing CnC Steps

Before a CnC step instance is executed, that step must be *prescribed* (created) and all of its input data items must be available. The CnC runtime tracks the status of step and item instances via *attributes* attached to the instances. When some step

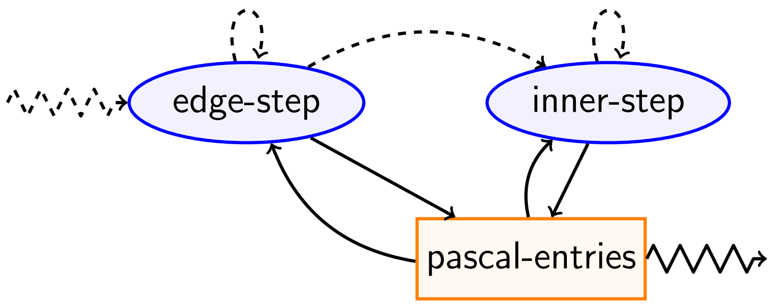
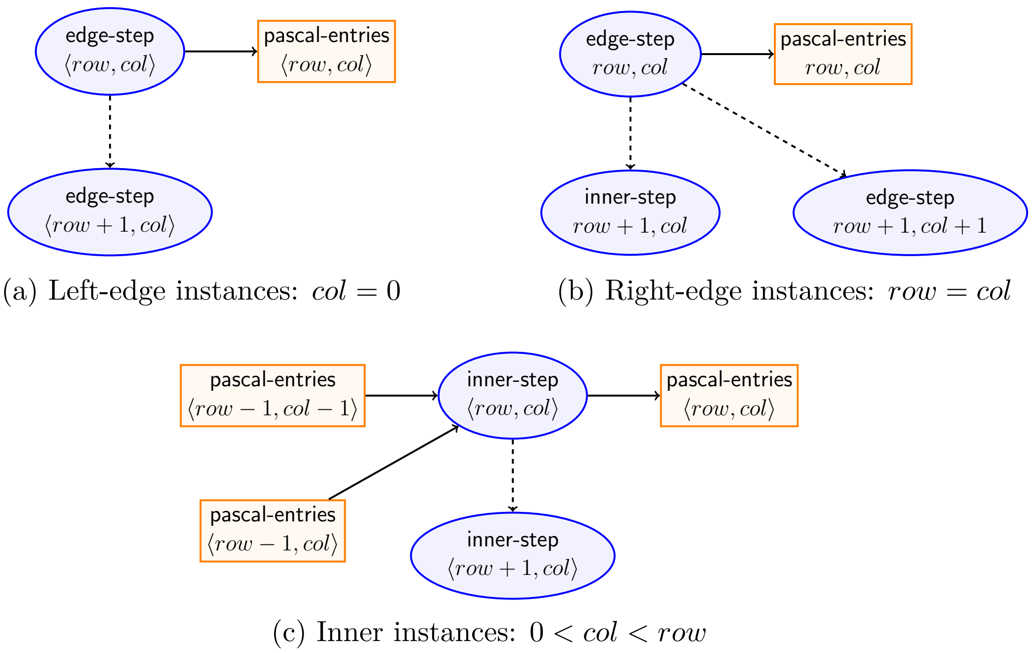


Figure 2.3 : The abstract graph representation of the Pascal's Triangle CnC application. Ellipses represent step collections (computation), and rectangles represent item collections (data). Dashed edges represent step prescriptions (creation), and solid edges represent puts to or gets from item collections. Jagged edges represent interactions with the application environment that encloses this CnC graph.


edge-step ⟨row, col⟩
pascal-entries ⟨row, col⟩
edge-step ⟨row + 1, col⟩
edge-step row, col
pascal-entries row, col
inner-step row + 1, col
edge-step row + 1, col + 1
pascal-entries ⟨row − 1, col − 1⟩
inner-step ⟨row, col⟩
pascal-entries ⟨row, col⟩
pascal-entries ⟨row − 1, col⟩
inner-step ⟨row + 1, col⟩

(a) Left-edge instances: $col = 0$

(b) Right-edge instances: $row = col$

(c) Inner instances: $0 < col < row$

Figure 2.4 : The *prescribe*, *put* and *get* relationships among the step and item collections in the Pascal's Triangle CnC application. Note that the topmost entry of the triangle, where $row = col = 0$, is actually a special case that does the *edge-step* prescription from the left edge and the *edge-step* prescription from the right edge without an *inner-step* prescription.

instance (or the environment) *prescribed* a step in collection S with tag T, step $(\!|\,\mathsf{S}\colon \mathsf{T}\,|\!)$ is created with the *control ready* attribute. If a step has a single input dependence on $[\![\,\mathsf{I}\colon \mathsf{K}\,]\!]$, the step gains the *data ready* attribute when an item with key K has been put to item collection I. If a step has two or more such dependencies, the step is data ready only when all of the input items have been put. If a step has zero input dependencies then it is always considered data ready. Once a step is both control ready and data ready, it gains the *ready* attribute, and is only then eligible to execute.

In our Pascal's Triangle application, an instance of *edge-step* is ready as soon as it is prescribed because it has no input dependencies on the item collection. Since instances of *inner-step* depend on two item instances from *pascal-entries*, an *inner-step* instance is only ready to execute after it has been prescribed and both of the corresponding item instances have been put.

Interaction with the Environment

A CnC graph is typically embedded within a driver application, and we refer to the portions of the application that interact with the CnC graph as the *environment*. When CnC program execution is completed, the environment must put all data, prescribe all steps, and set any parameters necessary to properly initialize the CnC graph. The environment may also get values from item collections, which acts as an output mechanism for the graph.

In our Pascal's Triangle application, the environment initializes the graph's n and k parameters, then prescribes an $(\!|\,\mathsf{edge\text{-}step}\colon 0{,}0\,|\!)$, which corresponds to the topmost entry of the triangle. From that point, the CnC runtime has all the information it needs to compute the value for ${}_nC_k$. When the CnC graph has completed its execution, the environment gets ⟦ pascal-entries: n,k ⟧, which holds the computed value of ${}_nC_k$.

Example Execution

We will now outline an example of an execution trace for our Pascal's Triangle application. For simplicity in tracing the execution, we assume that the runtime has only a single worker thread, meaning that only one step can run at a time. This assumption eliminates any possible concurrency among steps in the computation and simplifies reasoning about program execution and execution trace creation.

We pick ${}_2C_1$ as the target value for this execution, therefore the environment initializes an instance of our CnC graph with the parameters $n = 2$ and $k = 1$. The environment also prescribes ⦇ edge-step: 0,0 ⦈ to start the graph's execution. Since ⦇ edge-step: 0,0 ⦈ has been prescribed and has no input dependencies, it is ready to execute. The graph execution is described textually below, and graphically in figure 2.5.

⦇ edge-step: 0,0 ⦈
puts ⟦ pascal-entries: 0,0→1 ⟧;
prescribes ⦇ edge-step: 1,0 ⦈ and ⦇ edge-step: 1,1 ⦈.

All steps in row 0 have now run to completion.

⦇ edge-step: 1,0 ⦈
puts ⟦ pascal-entries: 1,0→1 ⟧;
prescribes ⦇ edge-step: 2,0 ⦈.

⦇ edge-step: 1,1 ⦈
puts ⟦ pascal-entries: 1,1→1 ⟧;
prescribes ⦇ inner-step: 2,1 ⦈ and ⦇ edge-step: 2,2 ⦈.

All steps in row 1 have now run to completion.

⦇ edge-step: 2,0 ⦈
puts ⟦ pascal-entries: 2,0→1 ⟧;
prescribes no steps since $row = n = 2$.

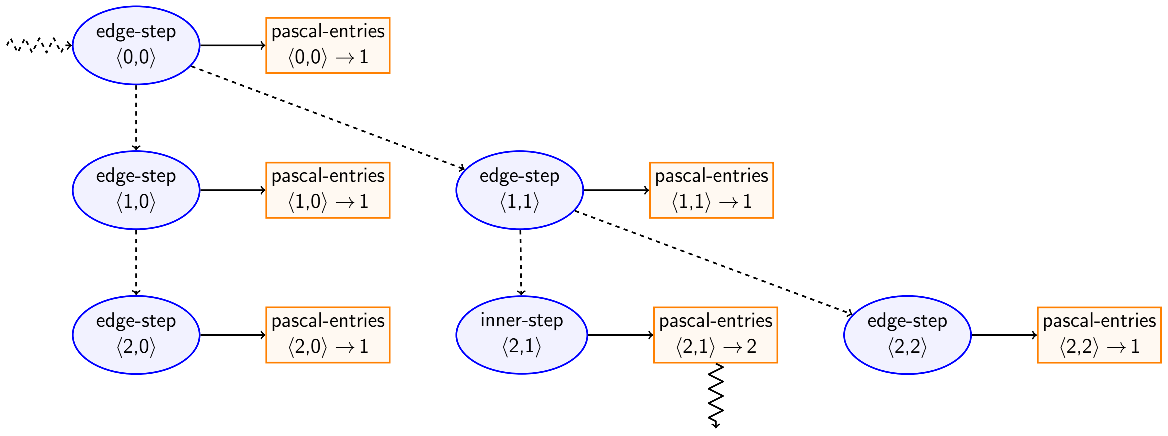


Figure 2.5 : Dynamic CnC graph for the computation of ${}_2C_1$.

⦇ inner-step: 2,1 ⦈ depends on ⟦ pascal-entries: 1,0 ⟧ and ⟦ pascal-entries: 1,1 ⟧, but since both items were already put, it is ready to execute.

⦇ inner-step: 2,1 ⦈
gets ⟦ pascal-entries: 1,0⟶1 ⟧ and ⟦ pascal-entries: 1,1⟶1 ⟧;
puts ⟦ pascal-entries: 2,1⟶2 ⟧;
prescribes no steps since $row = n = 2$.

⦇ edge-step: 2,2 ⦈
puts ⟦ pascal-entries: 2,2⟶1 ⟧;
prescribes no steps since $row = n = 2$.

All steps in row 2 have now run to completion. Since all prescribed steps have run to completion, the CnC graph execution is finished. The environment gets item instance ⟦ pascal-entries: 2,1⟶2 ⟧ and correctly yields the answer ${}_2C_1 = 2$.

2.1.3 The CnC Continuum

CnC describes a programming paradigm rather than a specific runtime implementation. As a result, there is quite a bit of flexibility in how a particular CnC runtime may behave, and what requirements it might impose. One example of this is the static or dynamic nature of the CnC graph. CnC has no restrictions about how much of a application's graph structure must be computable statically versus computed dynamically at runtime. This results in a variety of requirements in the existing CnC implementations pertaining to the specification of inputs and outputs of CnC step functions. Some implementations require that some or all of step tags and item keys to be computed statically, whereas others allow all the inputs and outputs of a step instance to be computed dynamically. The specifics of the particular flavor of CnC used in this thesis are described in section 3.1.

2.2 Current Approaches to Resilience

Current approaches to resilience fit into three major categories: kernel-level checkpointing, user-level checkpointing, and non-checkpoint-based resilience. We discuss the trade-offs of these three approaches to resilience throughout the rest of this section.

2.2.1 Kernel-Level Checkpointing

Kernel-level checkpointing involves modifying the operating system kernel to save snapshots of memory over time for use as checkpoints in case of a failure. An example of a kernel-level checkpointing system is Berkeley Lab Checkpoint Restart (BLCR) [11]. BLCR makes system-wide checkpoints, saving the state of all processes running on the current machine in order to restore the entire system's state in the event of a failure. This approach is advantageous in multiprogramming environments because all users' applications running on a given compute node are checkpointed together, rather than each incurring an individual overhead for C/R. This process is also generic enough that it will work for any running application, and BLCR even has support for cooperating with common messaging libraries for coordinating (synchronizing) inter-node checkpoints. However, this generality comes at the cost of an increased memory footprint since the lack of application-specific details necessitates saving the entire process image of each running application for the checkpoint. Additionally, kernel-level checkpoints require modification to the underlying operating system kernel in order to operate, meaning that only system administrators can install kernel-level C/R systems like BLCR.

2.2.2 User-Level Checkpointing

The user-level approach works within an application to provide C/R capabilities to that individual application. Applying C/R at the application level is more popular than system-level C/R because application-specific knowledge can be leveraged to decrease the overall I/O overhead and memory footprint [11].

The Chandy-Lamport algorithm [12] is a seminal work for checkpointing distributed systems. The algorithm provides a method to capture a consistent snapshot of the global state of a system without requiring global synchronization. Instead, each process asynchronously saves its own state, and then logs messages received in order to create the globally consistent snapshot. The Chandy-Lamport algorithm assumes a set of processes connected by a set of unidirectional message channels. The CnC model has no concept of communication channels; rather, all data is shared through the contents of the item collections. In this thesis we take advantage of the monotonically increasing state of CnC applications to create a more specialized checkpointing solution.

Charm++ is a variant of the popular C++ programming language that provides transparent C/R as part of the programming model [3]. Charm++ leverages additional information gained from runtime integration in order to reduce checkpoint memory footprints. We take a similar approach in reducing the memory footprint of checkpoints by saving only live application data encapsulated within the CnC runtime; however, we are also able to leverage CnC's semantics to avoid synchronization when saving and restoring checkpoints.

Distributed MultiThreaded CheckPointing (DMTCP) is a user-level checkpointing solution that works transparently (meaning that the application writer does not need to modify the application to accommodate C/R) for a large variety of applications [13]. DMTCP provides an interesting feature: it enables a user to migrate a checkpoint from a supercomputer to a desktop machine, and then restart from the checkpoint on the desktop machine to provide a better debugging experience. However, DMTCP uses global synchronization (a series of barriers) to coordinate checkpoints, which will not scale to provide acceptable performance on exascale systems. We provide migratable checkpoints while eliminating global synchronization.

2.2.3 Non-Checkpoint-Based Resilience

Although C/R tends to dominate resilience in the high-performance computing domain, there are other possible approaches. One example is the data-driven fault tolerance system designed by Ma and Krishnamoorthy [14]. Their system mirrors the computation state in an idempotent data store as data is updated, rather than periodically taking a snapshot of computation state as done traditionally in C/R. We also take a continuous-update approach in our solution, thus also varying from the traditional C/R model.

2.2.4 Previous C/R Work for CnC

HP labs implemented a prototype of C/R in CnC (known as TStreams at that time) [15], principally designed by Alex Nelson. Aside from a demonstration at the HP booth at Supercomputing, the details of this work were never published or otherwise made public, nor was the prototype made available outside HP in any other form. The C/R prototype was demonstrated with a single application—a graphical Mandelbrot set code. This thesis lays a theoretical foundation for C/R in CnC and aims to be more general and complete then the previous work done at HP Labs—although any comparison of generality or completeness is impossible since the original prototype is not available.

2.3 Habanero-C CnC

Habanero-C (HC) is an extension to the C language that adds constructs for parallel programming with *async/finish*-based parallelism [16, 17]. In CnC-HC, steps are scheduled using data-driven tasks [18]. We discuss the process of adding checkpoint/restart to the existing CnC-HC runtime in chapter 8.

The CnC-HC programming system includes a graph translator that takes in a textual representation of a CnC graph, then generates skeleton code for the application environment and all of the CnC step functions described by the graph, as well as generating scaffolding code to automatically get the inputs of all steps from the item collections based on the relationships described in the graph. The textual graph representation describes the relationships among the step and item collections using parameterized tag expressions, much like those shown in figure 2.4.

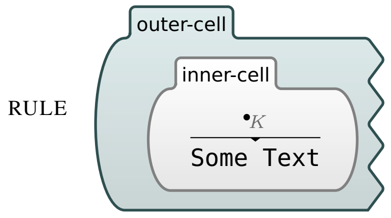


Figure 2.6 : Example of the graphical representation for a K rewrite rule. This rule adds “Some Text” to an empty ⟨*inner-cell*⟩ contained inside some ⟨*outer-cell*⟩. The torn edge on the ⟨*outer-cell*⟩ implies that it might contain other values as well, but this rule is only concerned with the portion shown.

2.4 Rewrite Rules and the K Framework

The K framework [19] is a system for building rewrite-rule based models. In this thesis we use the K-framework to model the behavior of the CnC runtime and the checkpointing process. A model specified in K consists of three pieces: a configuration, syntax, and rewrite rules. The state of a K model is represented by a string, which is divided up into a nested structure of *cells*. Since the K syntax uses an XML-like format for specifying the contents of cells, we use the notation ⟨foo⟩ to indicate a K cell with the name foo. Cells can also contain other text. The types of strings allowed as values in cells are determined by the syntax.

K allows for organizing syntax productions into groups called *sorts*—an abstraction similar to data types. K comes with some pre-declared sorts, such as *Ints*, *Bools*, *Lists* and *Bags*. The K sort is the top-sort that encompasses all strings. The • symbol is used to specify an empty string, and it can be annotated with a sort. Finally, the rewrite rules match cells and strings within the cells, and rewrite some portion of the match to a new value. A rule can match all or part of the contents of a given cell when doing a rewrite. When only a portion of a cell is matched in a rule, this is represented graphically by showing a *torn edge* on one side of the cell. When a new cell is being created and uses this partial match representation, it indicates that all contents not explicitly shown default to the values given in the configuration. Cell names in the configuration can end with the symbol * to indicate that zero or more copies might exist (defaulting to zero). Figure 2.6 is an example of a simple rewrite rule in K.

Models written in K are executable. They can read a static program as input, as well as dynamically interact with I/O through *stdin* and *stdout*.

Chapter 3

A Thin Wrapper for the CnC Runtime

In this chapter we introduce a thin wrapper over the CnC runtime, which constitutes a simple API that we can use to write CnC applications. This is a simple implementation in Clojure [20] (an implementation of Lisp that interoperates with standard Java libraries and runs on the JVM). We use promises to asynchronously communicate with the CnC runtime model, similar to how the Habanero flavors of CnC use data-driven futures [18] to maintain data dependencies. We introduce the underlying runtime model in chapter 4.

3.1 CnC Flavor

As described in section 2.1.3, there are a wide variety of flavors of CnC. In this section, we detail the assumptions about the flavor of CnC we assume both for this wrapper and for our models.

3.1.1 Asynchronous Communication

We assume that the calls between the CnC runtime and user code are asynchronous. We assume that all messages will eventually be delivered, but we make no assumptions about message order.

3.1.2 Input and Output of Steps

In our CnC model, we assume that inputs are *static*, but outputs are *dynamic*. Requiring *static inputs* means that the set of data items needed to run a given computation step must be *statically computable* from an explicit *tag function*. In other words, the programmer must provide a function for each step collection that, given only the tag corresponding to a specific step instance, can statically compute the set of item keys (each associated with a specific item collection) enumerating the inputs for that step. Having an explicit tag function simplifies the process of determining when a step becomes *data-ready*, and also matches the explicit tag functions used in the graph specifications of the Habanero CnC implementations [21].

Allowing *dynamic outputs* means that the outputs of a given step (including both item puts and step prescriptions) may be data-dependent. In other words, the quantity and identities of the items put and steps prescribed by a step can be a function of the run-time values of that step's inputs.

3.1.3 Data Representation Restrictions

Due to limitations in parsing arbitrary input to the K framework [22], we restrict data in our model to integers. This restriction is purely a product of the lack of support for defining parse rules on runtime input to the model. Similarly, step tags and item keys are restricted to non-empty integer tuples. This restriction places a major limit on the variety of applications that we can directly run against our model. However, these restrictions do not limit our ability to reason about CnC. The encoding details of keys, tags and data are inconsequential as long as they can be read, written and compared for equality. As supporting more complex data types would add significant complexity to the already complicated mechanisms, we feel that this is an acceptable compromise.

3.1.4 Dynamic Graph Restrictions

A valid CnC application must have an acyclic dynamic structure. We do not assume that the CnC runtime checks for duplicate step prescriptions. If the CnC runtime neither checks for duplicate step prescriptions nor prevents duplicate steps from becoming *enabled* and executing, then the application diverges in the event of a cycle (i.e., when the same step instance is prescribed twice). Restricting the application to generate an acyclic graph structure guarantees that the graph execution will eventually terminate, assuming that no individual step diverges and that the steps have a finite tag-space.

3.2 The API

We now define the user API for writing CnC applications that can be executed with our runtime. Sections 3.2.1 and 3.2.2 contain two full examples of using this API to write a CnC application. The API consists of the following macros and functions:

> `(defstepfn` ***`name context-binding tag-binding input-bindings`*** `&` ***`body`***`)`
>
> Macro for defining a *step function* for the CnC graph. The *input-bindings* is a vector of triplets. Each triplet has form `(`***`binding-sym collection-id tag-exp`***`)`. The *tag-exp* may return either an individual tag, or a collection of tags. When the step function is run, each *binding-sym* will be bound to the result of getting the specified item(s) from the item collection with the given *collection-id*. Similarly, the current graph context is bound to *context-binding*, and the step's tag to *tag-binding*.

`(cnc-run` ***`graph-spec init-fn result-fn`***`)`

Function for creating and executing a CnC graph. The *graph-spec* is a map defining the step collections, item collections and any static data in the graph. The `:stepcolls` key specifies a mapping of collection-id/step-function pairs. The `:itemcolls` key specifies a mapping of collection-id/get-function pairs (get functions will be introduced in section 6.1.2). The optional `:static` key specifies a mapping of keywords to static values. The *init-fn* and *result-fn* are, respectively, functions for initializing the graph and returning a result from the graph data. Each takes a single parameter corresponding to the current graph context.

`(put-item` ***`context collection-id key value`***`)`

Creates an instance in the specified item collection with the given key and value.

`(prescribe-step` ***`context collection-id tag`***`)`

Creates an instance of a step from the specified collection with the given tag.

`(get-static` ***`context key`***`)`

Retrieves the static value from the graph context associated with the given key.

`(get-item` ***`context collection-id key`***`)`

Retrieves the value of the item in the specified collection with the given key. This is used implicitly to create the input bindings for step functions, but may also be used explicitly in the `cnc-run` function's *result-fn*.

3.2.1 Example: Pascal's Triangle

Listing 3.1 contains the full source code of a CnC application for computing binomial coefficients. The value ${}_nC_k$ is found by computing the n^{th} row and k^{th} column

```
;; Namespace for Pascal's Triangle application
(ns cnc.examples.pascal
  ; Import CnC API
  (:use cnc.runtime.core))

;; Unique IDs for each CnC collection
(def pascal-entries 0)
(def edge-step-id   1)
(def inner-step-id  2)

;; Step producing edge entries of the triangle
;;   ctx : current CnC context
;;   [row col] : tag specifying a row and column in the triangle
;;   [] : no data items are required as input
(defstepfn edge-step ctx [row col]
  []
  (assert (or (== col 0) (== col row)) "Must be an edge entry")
  ; Put the value 1 for the entry at (row, col)
  (put-item ctx pascal-entries [row col] 1)
  ; All but last row will prescribe a step in the next row
  (when (< row (get-static ctx :n))
    (let [step-id (if (zero? col) edge-step-id inner-step-id)]
      (prescribe-step ctx step-id [(inc row) col]))
    ; Right-edge entries also prescribe last entry in the next row
    (when (== row col)
      (prescribe-step ctx edge-step-id [(inc row) (inc col)]))))

;; Step producing inner entries of the triangle
;;   ctx : current CnC context
;;   [row col] : tag specifying a row and column in the triangle
;;   [x y] : values of entries at (row-1, col) and (row-1, col-1)
(defstepfn inner-step ctx [row col]
  [(x pascal-entries [(dec row) (dec col)])
   (y pascal-entries [(dec row) col])]
  (assert (< 0 col row) "Must be an inner entry")
  ; Put the value for the entry at (row, col) by
  ; adding the values x and y from the previous row
  (put-item ctx pascal-entries [row col] (+ x y))
  ; All but last row will prescribe a step in the next row
  (when (< row (get-static ctx :n))
    (prescribe-step ctx inner-step-id [(inc row) col])))

```

Listing continued on next page.

Listing continued from previous page.

```
;; Get count function for the pascal-entries collection
(defn entry-get-count [ctx [row col]]
  (let [n (get-static ctx :n)
        k (get-static ctx :k)]
    (cond
      ; All entries in the last row are never read,
      ; except the kth entry, which is read once as output
      (== row n) (if (== col k) 1 0)
      ; Row 0 is not read
      (== row 0) 0
      ; Edge entries are only read once
      (or (== k 0) (== k n)) 1
      ; Other entries are read twice
      :else 2)))

;; Compute n choose k by computing n rows of Pascal's Triangle.
;; This function provides the environment for the CnC graph.
(defn n-choose-k [n k]
  (cnc-run
    ; CnC graph specification: maps step collection IDs
    ; to step functions, item collection IDs to get-count
    ; functions, and setting values for static parameters.
    {:stepcolls {edge-step-id   edge-step
                 inner-step-id  inner-step}
     :itemcolls {pascal-entries entry-get-count}
     :static    {:n n, :k k}}
    ; CnC graph initialization function
    ; Start a step for the entry at the top of the triangle
    (fn [ctx]
      (prescribe-step ctx edge-step-id [0 0]))
    ; CnC graph result-processing function
    ; Return the entry from row n, column k of the triangle
    (fn [ctx]
      (get-item ctx pascal-entries [n k]))))
```

Listing 3.1: Clojure code for the Pascal's Triangle CnC application.

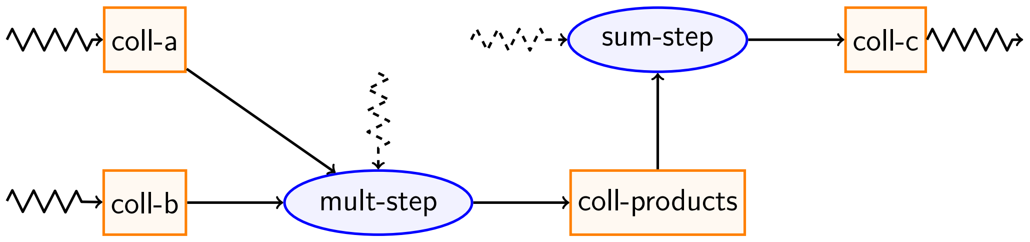


Figure 3.1 : The abstract graph representation of the matrix multiplication CnC application. Ellipses represent step collections (computation), and rectangles represent item collections (data). Dashed edges represent step prescriptions, and solid edges represent puts to or gets from item collections. Jagged edges represent interactions with the application environment that encloses this CnC graph.

of Pascal's Triangle. Section 2.1.2 contains a discussion of the structure and behavior of this application, and its graph structure is illustrated in figure 2.3. The function `entry-get-count` on line 43 is explained in section 6.1.2. The following is a sample call and result for the application's `n-choose-k` function:

```
(n-choose-k 4 2) ; result: 6
```

3.2.2 Example: Matrix Multiplication

Listing 3.2 contains the full source code of a CnC application for computing the product of two matrices. The structure of the application is illustrated in figure 3.1. We now give a brief outline of the structure and behavior of this application. The following is a sample call and result of the application's `matrix-mult` function:

```
(matrix-mult [[1] [2]] [[3 4]]) ; result: [[3 4] [6 8]]
```

Since we specify a matrix as a nested vector in this application, the function call and result in the code fragment above is equivalent to the following equality:

```clojure
;; Namespace for Matrix Multipy application
(ns cnc.examples.matrix-multiply
  ; Import CnC API
  (:use cnc.runtime.core))

;; Unique IDs for each CnC collection
(def coll-a        0) ; items for input matrix a
(def coll-b        1) ; items for input matrix b
(def coll-c        2) ; items for output matrix c
(def coll-products 3) ; items for intermediate products
(def mult-id       4) ; steps to multiply entries
(def sum-id        5) ; steps to sum intermediate products

;; Step producing the intermediate products of the entry in
;; a-row, column i of matrix a and row i, b-col of matrix b.
;;   ctx : current CnC context
;;   [a-row b-col i] : tag specifying pair of entries to multiply
;;   [a-entry b-entry] : pair of entries from matrix a and b
(defstepfn mult-step ctx [a-row b-col i]
  [(a-entry coll-a [a-row i])
   (b-entry coll-b [i b-col])]
  ; Multiply the entries from matrices a and b and put the result
  (let [product (* a-entry b-entry)]
    (put-item ctx coll-products [a-row b-col i] product)))

;; Step summing the intermediate products from a-row of
;; matrix a and b-col of matrix b, producing the entry
;; at a-row, b-col of matrix c.
;;   ctx : current CnC context
;;   [a-row b-col] : tag specifying sequence of entries to multiply
;;   [ps] : sequence of products to sum, completing a dot product
(defstepfn sum-step ctx [a-row b-col]
  [(ps coll-products
       ; Get a sequence of products as input
       (let [indices (range (get-static ctx :a-cols))]
         (for [i indices] [a-row b-col i])))]
  ; Sum all of the input products
  (let [sum (reduce + ps)]
    ; Put the result to (a-row, b-col) in output matrix collection
    (put-item ctx coll-c [a-row b-col] sum)))

```

Listing continued on next page.

Listing continued from previous page.

```
;; Multiply a * b and return the resulting matrix c.
;; Matricies are input as vectors of vectors of integers.
;; This function provides the environment for the CnC graph.
(defn matrix-mult [a b]
  ; Calculate the dimensions of the input matrices
  (let [a-rows (count a)
        a-cols (count (a 0))
        b-rows a-cols
        b-cols (count (b 0))]
    (cnc-run
      ; CnC graph specification: maps step collection IDs
      ; to step functions, item collection IDs to get-count
      ; functions, and setting values for static parameters.
      {:stepcolls {mult-id       mult-step
                   sum-id        sum-step}
       :itemcolls {coll-a        (constantly b-cols)
                   coll-b        (constantly a-rows)
                   coll-c        (constantly 1)
                   coll-products (constantly 1)}
       :static    {:a-rows a-rows, :a-cols a-cols,
                   :b-rows b-rows, :b-cols b-cols}}
      ; CnC graph initialization function
      (fn [ctx]
        ; Put all entry values for matrix a into coll-a,
        ; and for matrix b into coll-b.
        (doseq [[id m] {coll-a a, coll-b b}
                r (range (count m))
                :let [row (m r)]
                c (range (count row))
                :let [entry (row c)]]
          (put-item ctx id [r c] entry))
        ; Prescribe all steps for multiplying corresponding
        ; entries in matrices a and b, then summing the products
        ; to compute the values of the output matrix c.
        (doseq [r (range a-rows)
                c (range b-cols)]
          (doseq [i (range a-cols)]
            (prescribe-step ctx mult-id [r c i]))
          (prescribe-step ctx sum-id [r c])))
      ; CnC graph result-processing function
      ; Collect and return entries of the output
      ; matrix c as a vector of vectors of integers.
      (fn [ctx]
        (vec (for [r (range a-rows)]
               (vec (for [c (range b-cols)]
                      (get-item ctx coll-c [r c])))))))))
```

Listing 3.2: Clojure code for the matrix multiplication CnC application.

$$\begin{bmatrix}1\\2\end{bmatrix} \times \begin{bmatrix}3 & 4\end{bmatrix} = \begin{bmatrix}3 & 4\\6 & 8\end{bmatrix}$$

The function `matrix-mult`, which acts as the environment for the CnC graph, takes two input matrices, `a` and `b`, and computes their matrix product using CnC. The CnC graph takes the dimensions of the two input matrices as parameters, although `a-cols` (the number of columns in matrix `a`) is the only parameter actually used in the step code. The other dimensions of the matrices are used by the environment to set up the graph, and by the get count functions. The get count functions (shown in the CnC graph specification on lines 57–60) are explained in section 6.1.2.

The graph has three item collections corresponding to the three matrices: `coll-a` and `coll-b` for the input matrices `a` and `b`, and `coll-c` for the output matrix `c`. Each matrix item collection has a tag consisting of two integers, representing the row and column of an entry in the matrix. For example, we would expect to have an instance ⟦coll-c: 1,0→6⟧ by the end of the computation. At the beginning of the computation the environment puts each entry from both input matrices into the corresponding item collection (see lines 67–72).

There is one additional item collection, `coll-products`, for holding the intermediate products during the computation. For example, ⟦coll-products: 0,1,0⟧ will hold the product ⟦coll-a: 0,0⟧ × ⟦coll-b: 0,1⟧, and ⟦coll-products: 1,1,0⟧ will hold the product ⟦coll-a: 1,0⟧ × ⟦coll-b: 0,1⟧. These products are created by `mult-step`, and sequences of products ⟦coll-products: row,col,i⟧ are summed by `sum-step`.

The `sum-step` specifies a *sequence* of items as input (see lines 33–36). It gets as input the sequence of item instances of the form ⟦coll-products: row,col,i⟧, for $0 \leq$ `i` $<$ `a-cols`. The sum of that sequence of products is then put as ⟦coll-c: row,col⟧.

In this application, the environment prescribes all of the step instances at the beginning of the computation, based off the input matrix dimensions (see lines 76–80). Once all step instances have run to completion, the environment gets all of the item instances from `coll-c` to build and return the output matrix `c` (see lines 81–87).

3.3 Summary

In this chapter we introduced the API used to interface programs written in Clojure with our CnC model. We outlined our assumptions about the flavor of CnC used in our model. We also introduced two sample CnC applications, written against our API, which we will reference throughout the remainder of this thesis. The full source code for the wrapper API and the two example applications introduced in this chapter are available at http://habanero.rice.edu/vrvilo-ms.

Chapter 4

A Formal Model of the CnC Runtime

In this chapter, we introduce our formal model of a runtime for the CnC programming paradigm. This runtime model serves as the basis for our checkpoint/restart system described in chapter 5. This initial runtime model takes into account no optimizations, focusing only on the basic semantics of CnC. We later expand on this model in chapter 6.

4.1 Building Executable Models

Models built within the K framework are *executable*. You can pass an input program to the model at startup, and continue to interact with it dynamically through *stdin* and *stdout* as it runs. This is a useful property because it allows us to perform a sanity check on the correctness of our model. While running test applications against the model does not prove its correctness, it does give us a certain level of confidence when the model gives the expected results on a small set of test runs.

Since our model is executable, we can use it as the back-end for the thin CnC wrapper API introduced in chapter 3.

4.2 The Runtime Model

In this section, we introduce the details of our K-based CnC model. (See section 2.4 for background information on the K framework.)

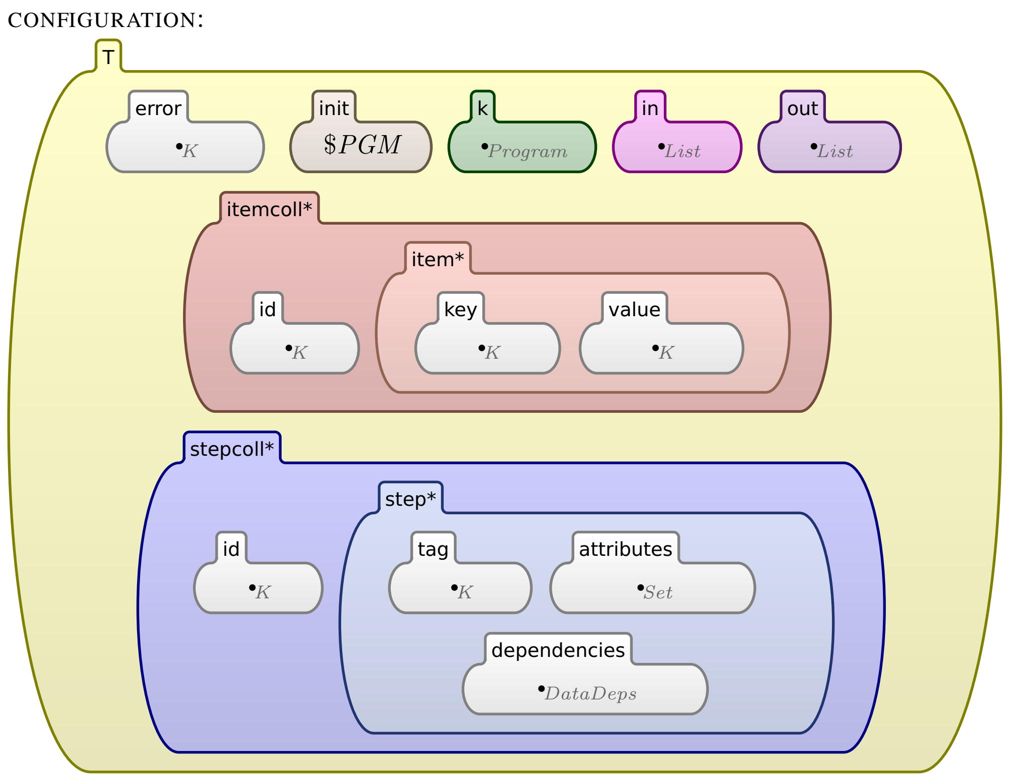


Figure 4.1 : K configuration for an unoptimized CnC runtime.

4.2.1 Configuration

Figure 4.1 illustrates the K configuration of our unoptimized CnC runtime model. The configuration encapsulates the complete state of the model. We describe the functions of the individual cells in this section.

⟨***error***⟩ Holds an error message in the case that the runtime crashes. The default value is $\bullet K$ (empty) because the model starts in an non-error state.

⟨***init***⟩ Holds the graph initialization commands. The special $\$PGM$ variable must contain an `item-coll` or `step-coll` command to initialize each item collection and step collection in the graph.

⟨***k***⟩ The computation cell. Holds the commands for manipulating the model state. Empty by default, but the contents of ⟨*init*⟩ and ⟨*in*⟩ will be used to add new commands to this cell.

⟨***in***⟩ New commands are read through this cell during model execution. Initialized to an empty list by default; however, the contents of *stdin* are automatically added to the list during model execution.

⟨***out***⟩ Output is given through this cell during execution. Items added to the cell are written to *stdout* during model execution. Defaults to the empty list since the model starts with no output.

⟨***itemcoll****⟩ Each represents one item collection in this graph.

- ⟨***id***⟩ Holds the unique ID of this item collection.
- ⟨***item****⟩ Each represents one item in this collection.
  - ⟨***key***⟩ Holds the unique key identifying this item.
  - ⟨***value***⟩ Holds this item's value.

⟨***stepcoll****⟩ Each represents one step collection in this graph.

- ⟨***id***⟩ Holds the unique ID of this step collection.
- ⟨***step****⟩ Each represents one step in this collection.
  - ⟨***tag***⟩ Holds the unique tag identifying this step.
  - ⟨***attributes***⟩ Holds the set of CnC attributes describing this step.
  - ⟨***dependencies***⟩ Holds a list of unique identifiers (item collection ID + item key) corresponding to all input data dependencies of this step.

It is interesting to note that almost the entire CnC runtime state is encapsulated in $\langle itemcoll{*}\rangle$ (the item collections) and $\langle stepcoll{*}\rangle$ (the step collections). If the $\langle init\rangle$ cell is non-empty, then the graph is not yet fully initialized, and the entire state is invalid. Likewise, if the $\langle error\rangle$ cell is non-empty, then the entire state is invalid, including the step and item collections. The rest of the cells in our configuration only exist to facilitate commands updating or querying $\langle itemcoll{*}\rangle$ and $\langle stepcoll{*}\rangle$, implying that $\langle k\rangle$, $\langle in\rangle$ and $\langle out\rangle$ can always be ignored when considering the current state of the CnC graph.

4.2.2 Syntax

Figure 4.2 outlines the syntax used in the rewrite rules of our model. Figure 4.2a defines the syntax of the input program used to initialize the model (denoted by $\$PGM$ in figure 4.1). It is simply a non-delimited list of `(item-coll` ID`)` and `(step-coll` ID`)` commands, where ID is some integer. Figure 4.2b enumerates the sorts of data that are acceptable as results in the $\langle k\rangle$ cell. This basically tells the rewrite engine which expressions cannot be further reduced.

Figure 4.2c defines the basic data types used in our model. As explained in section 3.1.3, we limit data to integers, and tags (which encompass both step tags and item keys) are limited to non-empty integer tuples. The distinction between a *DataItem* and a *DataExp* is that our I/O module extends *DataExp* to include commands for reading data from input. The same holds for the other **-Exp* variants of our syntactic sorts. Figure 4.2d defines *dependency* sorts, which combine a collection ID with a tag/key to uniquely identify a step/item instance in a particular collection. At the bottom of 4.2d is the list of possible *attributes* that can be associated with a step in our model. These attributes correspond to those described in section 2.1.2.

SYNTAX *CollID* ::= *Int*

SYNTAX *InitCommand* ::= (`item-coll` *CollID*)
| (`step-coll` *CollID*)

SYNTAX *InitSeq* ::= $List\{InitCommand, ``"\}$

(a) Graph initialization

SYNTAX *KResult* ::= *Halt*
| *Int*
| *Tag*
| *DataDeps*
| *DepPair*

(b) Computation result sorts

SYNTAX *IDExp* ::= *CollID*

SYNTAX *DataItem* ::= *Int*

SYNTAX *DataExp* ::= *DataItem*

SYNTAX *DataItems* ::= *DataItems DataItem*
| *DataItem*

SYNTAX *Tag* ::= (*DataItems*)

SYNTAX *TagExp* ::= *Tag*

(c) Data and tags

SYNTAX *DepPair* ::= *IDExp TagExp* [seqstrict]

SYNTAX *DepPairExp* ::= *DepPair*

SYNTAX *DataDeps* ::= $List\{DepPairExp, ``"\}$ [seqstrict]

SYNTAX *DepsExp* ::= *DataDeps*

SYNTAX *Attribute* ::= `control-ready`
| `data-ready`
| `enabled`
| `done`

(d) Dependency lists and step attributes

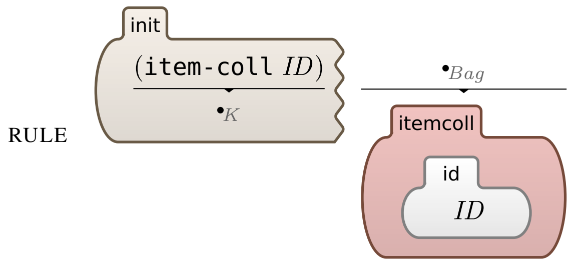
```
SYNTAX   Program ::= List{InCommand, “”}

SYNTAX   InCommand ::= (put IDExp  TagExp  DataExp) [seqstrict]
                     | (prescribe IDExp  TagExp) [seqstrict]
                     | (add-data-deps IDExp  TagExp  DepsExp) [seqstrict]
                     | (get IDExp  TagExp) [seqstrict]
                     | (step-done IDExp  TagExp) [seqstrict]
                     | Halt

SYNTAX   OutCommand ::= (query-data-deps CollID  Tag)
                      | (run-step CollID  Tag)
                      | (give CollID  Tag  DataItem)
                      | (finished)
                      | (error-thrown)

SYNTAX   Halt ::= halt
                | error-halt

SYNTAX   ReadCmd ::= (read-cmd)
```

(e) CnC API commands

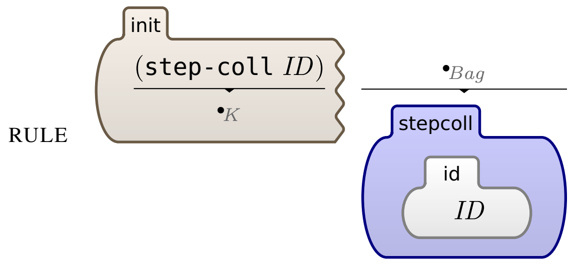
Figure 4.2 : Syntax of our unoptimized CnC model's rewrite rules.

Figure 4.2e outlines the set of commands use to interact with the CnC model. The *InCommand* sort describes all the commands that a user can send to the model, and the *OutCommand* sort enumerates the messages that the user can receive from the runtime. The `(finished)` code is output by the runtime model when all prescribed steps have run to completion. Once the environment is done getting any needed output data from the graph state, it sends the `halt` to the runtime model, signaling that the CnC graph is no longer needed and the runtime can shut down. Finally, `(read-cmd)` is a special command used to get a new *InCommand* from the input.

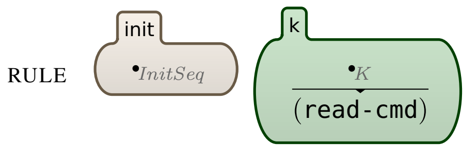
Several of the productions in figure 4.2 are annotated with *seqstrict*. Those productions are all compounds of multiple expressions, and this annotation specifies that the expressions must be evaluated sequentially, left-to-right. Without this annotation, the rewrite framework might choose to evaluate the expressions out-of-order,



(a) Create item collections



(b) Create step collections



(c) Read commands from input after initialization finishes

Rule Set 4.1: Rewrite rules for initializing all collections in the graph, and reading commands once the initialization is complete.

which would fail since the order of the expressions corresponds to the order they must appear in the input and output.

4.2.3 Rewrite Rules

In this section, we describe the rewrite rules of our unoptimized CnC runtime model. These rules in turn define the exact behavior of our model.

Initialization

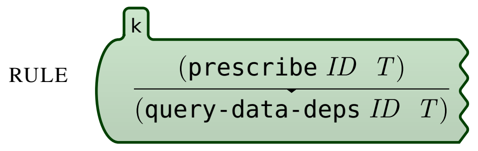
Rules 4.1a and 4.1b describe how item collections and step collections are created, respectively. In each case, an initialization command is read from the ⟨*init*⟩ cell, providing a unique ID. When the rule executes, the matched initialization command is deleted, while its ID is copied into the ⟨*id*⟩ cell of a newly created, empty ⟨*itemcoll*⟩ or ⟨*stepcoll*⟩. Rule 4.1c describes the constraint that a new rule is only read into the ⟨*k*⟩ cell for computation when initialization has finished (i.e., ⟨*init*⟩ is empty) and there is no other command currently executing (i.e., $\langle k \rangle$ is empty). The details of how `(read-cmd)` is rewritten into a new command from the input is covered in appendix B.

Input Commands

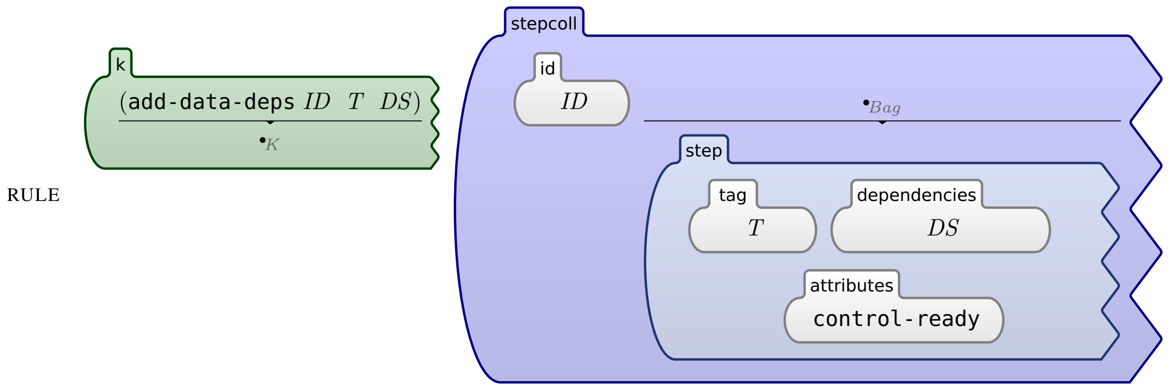
Rule sets 4.2 and 4.3 show the commands for updating and querying the item and step collections. The `get` (rule 4.3b) and `prescribe` (rule 4.2a) rules do not update the runtime state. Instead, `get` responds with the value of the requested item, and `prescribe` runs the tag function for the specified step to find its dependencies. The tag function should respond with a `add-data-deps` command (rule 4.2b), containing the list of input data dependencies for the step, which then creates a new step in the matching step collection using the provided data, and marks it as `control-ready`. The `put` command (rule 4.3a) similarly creates a new item. Once a step has run to completion, it can signal that with a `step-done` command (rule 4.2c), which adds the `done` attribute to the matching step.

Rule 4.3c enforces *dynamic single assignment* of items in CnC. If an item with a duplicate key is put in a collection, this should be an error when the two items' values do not match. If the two values match, then the two items are indistinguishable and it is equivalent to having only the original item in the collection.

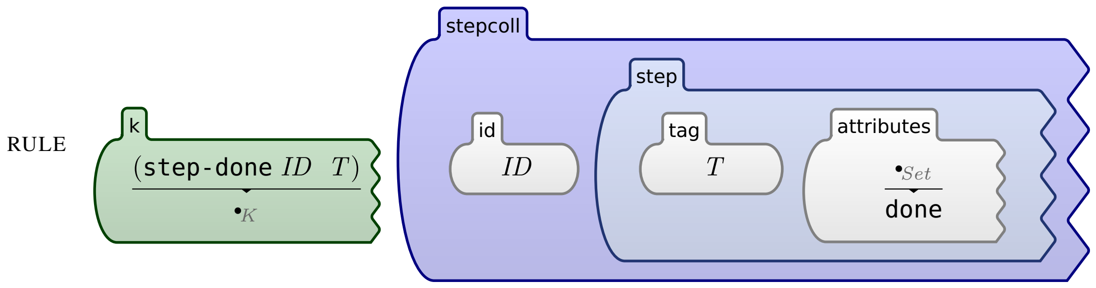
Note that we do not include a rewrite rule for *Halt* despite the fact that it falls under the productions for *InCommand*. This is because *Halt* is a result term (as described in section 4.2.2 and shown in figure 4.2b); therefore, when the `halt` command is encountered in the $\langle k \rangle$ cell, it means the computation is finished and no more rewrites are necessary.



(a) Prescribe a step

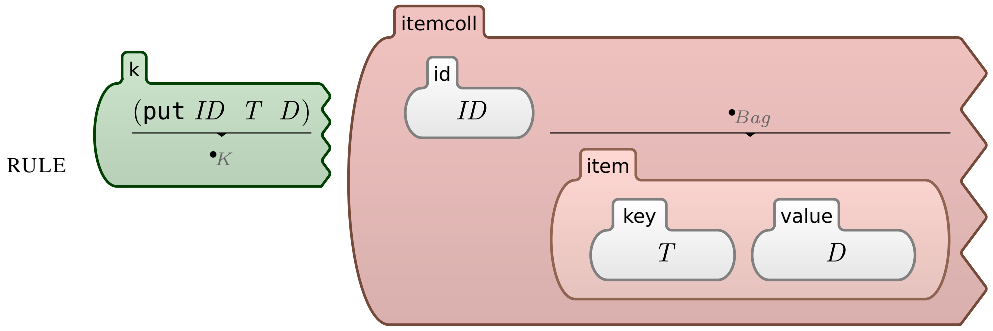


(b) Specify a step's data dependencies

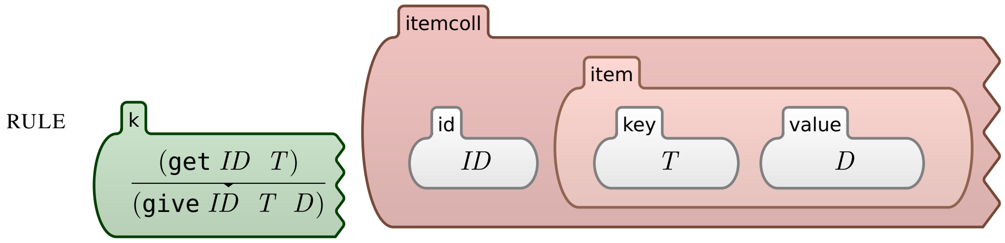


(c) Signal that a step has finished

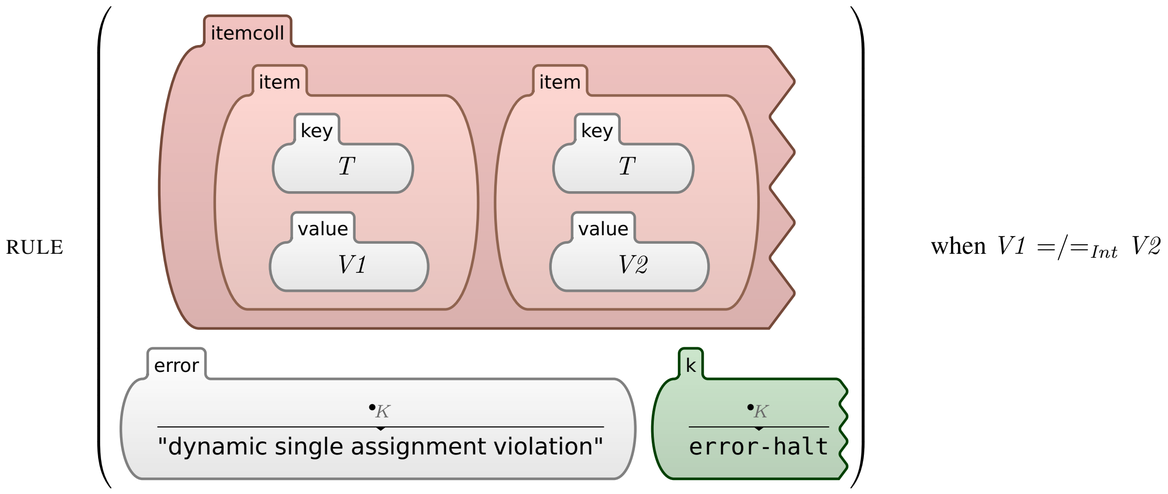
Rule Set 4.2: Rewrite rules for prescribing new steps, specifying their inputs, and signaling when a step's execution has completed.



(a) Put an item

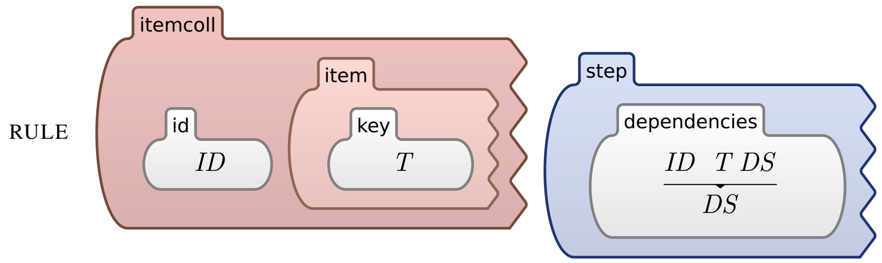


(b) Get an item

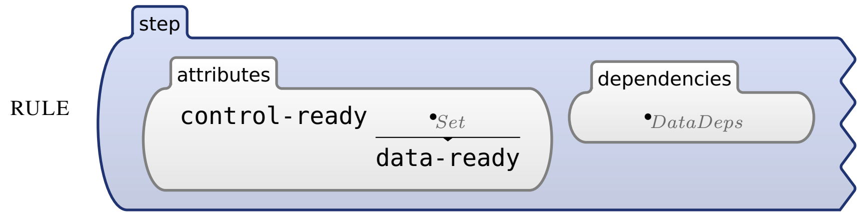


(c) Enforcing dynamic single assignment

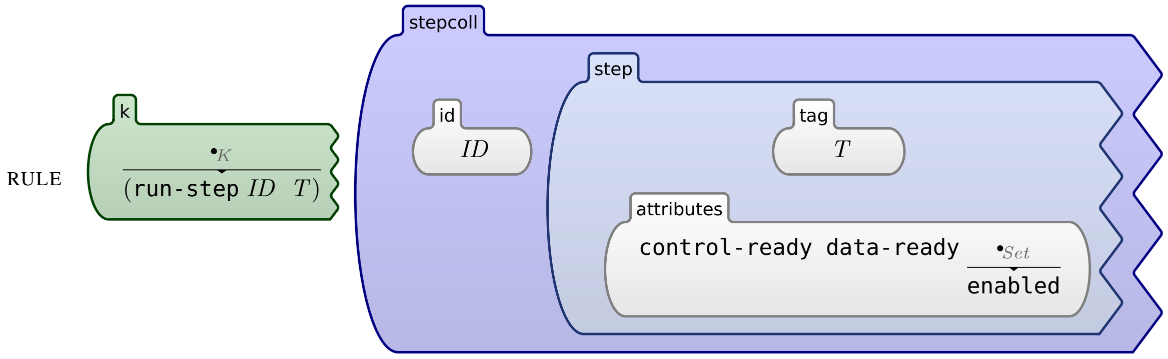
Rule Set 4.3: Rewrite rules for adding to and getting from item collections.



(a) Satisfy a data dependence



(b) Step is data ready


RULE
k
$\frac{\bullet_{K}}{(\texttt{run-step}\ ID\ \ T)}$
stepcoll
id
ID
step
tag
T
attributes
control-ready data-ready $\frac{\bullet_{Set}}{\texttt{enabled}}$

(c) Step is enabled

Rule Set 4.4: Rewrite rules for satisfying a step's data dependencies and running that step.

Steps and Dependencies

Rule set 4.4 shows the rules for satisfying a step's data dependencies and running that step. When the first dependence—represented as an item collection ID `+` item key pair—in a step's ‹*dependencies*› cell matches an existing item, that dependence is deleted, as shown in rule 4.4a. When ‹*dependencies*› is empty, the step is updated with the `data-ready` attribute, as shown in rule 4.4b. Now the step is both `control-ready` and `data-ready`, which means it is ready to be run. The step is then updated with the `enabled` attribute, and the `run-step` command is added to the ‹*k*› cell to run the corresponding step code, as shown in rule 4.4c. Note that while these last two rules could be combined into a single rule, we chose to keep them separate as we believe that the two smaller updates are easier to comprehend than a larger more complex update.

4.2.4 Handling I/O in the Executable Model

Our executable model supports dynamic I/O for interaction with a driver application via *stdin* and *stdout*. However, the details of this I/O process are beyond the scope of this thesis. The additional syntax and rewrite rules used to support I/O are presented in appendix B.

4.3 Key Properties of CnC

Now that we have a clear and concise definition of the CnC runtime, we can define five key properties of CnC and prove that they hold in the context of our model. These properties closely parallel those introduced in section 2.1.1, where we described how they simplify the C/R process for CnC applications. We will leverage these five key properties to implement C/R in chapter 5.

Theorem 4.1

CnC steps are idempotent.

Proof 4.1 Due to the single-assignment property of CnC items, a step's inputs remain constant throughout any given graph execution. Since all data produced by a step is a pure function of its input data, and that data is constant, re-running a given step can only result in identical output. Therefore, the set of key/value pairs for all items in all data collections is the same once all steps have finished regardless of the number of times any set of steps is re-run, or when they are re-run. Additionally, a partial run of a step will only produce some subset of the outputs of a complete run of a step; therefore, the result of any number of partial runs of a step with at least one complete run of that step is also idempotent. ■

Theorem 4.2

CnC graph state is monotonic.

Proof 4.2 Since there is no rewrite rule to add data to the $\langle init \rangle$ cell, its contents can only monotonically decrease via rules 4.1a and 4.1b. Once it becomes empty the graph state becomes valid, and the graph state cannot thereafter be invalidated by a non-empty $\langle init \rangle$. In other words, once the graph is initialized it stays initialized. Somewhat symmetrically, the $\langle error \rangle$ cell is only referenced by rule 4.3c, which adds an error string to the cell. Once this rules has been applied, execution cannot continue because $\langle k \rangle$ contains the `error-halt` value; therefore, $\langle error \rangle$ can be updated at most once and thus it increases monotonically. It also follows that once a graph is in an error state, it stays in that error state.

As noted in section 4.2.1, given that ⟨*error*⟩ and ⟨*init*⟩ are both empty, only the contents of the ⟨*itemcoll**⟩ and ⟨*stepcoll**⟩ cells are relevant to the CnC graph state. All rewrite rules monotonically increase the data in these cells, with the exception of rule 4.4a, which only updates the ⟨*dependencies*⟩ cell. The ⟨*dependencies*⟩ cell is the only cell in ⟨*itemcoll**⟩ or ⟨*stepcoll**⟩ from which data is deleted. However, as the only other rewrite rule that updates this cell is 4.2b, which initializes ⟨*dependencies*⟩ to a *DataDeps* list with zero or more entries, the data in this cell can only monotonically decrease.

Since all cells relevant to the state of the CnC graph are updated monotonically, we can therefore conclude that the entire state of the CnC graph is monotonic throughout execution. Furthermore, since the *DataDeps* list in ⟨*dependencies*⟩ and the $InitSeq$ list in ⟨*init*⟩ are not exposed through any CnC command, we can therefore conclude that the observable state of the CnC graph can only increase monotonically. Furthermore, since this argument is based entirely on the rewrite rules and the structure of the configuration cells, we need not make any special assumptions about repeated step executions. ■

Theorem 4.3

The CnC graph encapsulates all data necessary to complete its execution.

Proof 4.3 From the definition of the CnC programming paradigm, a tag function can only read static data, and steps may only obtain dynamic input from the graph's item collections. The CnC graph's item collections are seeded with initial data by the graph initialization function (provided by the environment); all other data items are generated by step code during the graph execution. Therefore, once the graph initialization function has completed, the CnC graph contains all the data necessary to complete its execution. Since the graph's items increase monotonically, the graph always encapsulates all of its necessary data to complete execution from any control state observed from the time it was initialized to the current state. ■

Theorem 4.4

The CnC graph encapsulates all control-flow information necessary to complete its execution.

Proof 4.4 The CnC graph's step collections are seeded with initial steps by the graph initialization function (provided by the environment); all other steps are prescribed by step code during the graph execution. Therefore, once the graph initialization function has completed, the CnC graph contains all control flow information necessary to complete its execution. Since the graph's items increase monotonically, the graph always encapsulates all of its necessary data to complete execution from the current and all previous control states. ■

Theorem 4.5

CnC graphs are deterministic, i.e., a graph given the same inputs will always produce the same outputs.

Proof 4.5 All CnC steps are idempotent pure functions. For a given input to the graph, the output of the initial steps is fixed. This applies transitively to all steps that work on the output of the initial steps, etc. This applies to all data produced by the graph, implying that the final graph state is a deterministic function of the initial input. Therefore, the CnC graph is deterministic. ■

4.4 Summary

We have defined the semantics of CnC in 12 rewrite rules. We also proved that five key properties of CnC (theorems 4.1 to 4.5) hold for our model. We will use these five properties in chapter 5 to construct a formal model for checkpoint/restart of CnC applications.

Chapter 5

A Formal Model for Checkpoint/Restart in an Unoptimized CnC Runtime

In this chapter, we introduce our formal model of the our CnC checkpointing system. This initial model is based on our model of the unoptimized CnC runtime, as defined in chapter 4. After adding various optimizations to the CnC runtime in chapter 6, we expand on this model in chapter 7.

5.1 The Checkpoint Model

5.1.1 Configuration

Figure 5.1 illustrates the K configuration of our unoptimized CnC checkpoint model. The configuration encapsulates the complete state of the model. The functions of the individual cells are the same as the cells with identical names described in section 4.2.1, with a few additions as follow:

⟨***x-key***⟩ Holds an *extended* key for the given item (item collection ID + item key pair).

⟨***x-tag***⟩ Holds an *extended* tag for the given step (step collection ID + step tag pair).

⟨***putter***⟩ Holds an extended tag corresponding to the step that put this item.

⟨***prescriber***⟩ Holds an extended tag corresponding to the step that prescribed this step.

⟨***step-log****⟩ Each represents the log of the put and prescribe activity of a given step.

⟨***puts-count***⟩ Holds an integer count representing the number of unsatisfied *put* operations done by the given step.

⟨***prescribes-count***⟩ Holds an integer count representing the number of unsatisfied *prescribe* operations done by the given step.

5.1.2 Syntax

The syntax for the checkpoint model's rewrite rules is almost identical to that of the general CnC runtime, as defined in section 4.2.2. The differences introduced for checkpointing are shown in figure 5.2. First, we add a unique extended tag value, `cnc-env`, which we use to identify the environment's input. This new extended tag is also added as a valid production of the *DepPair* sort. The `put` and `prescribe` commands now each read an additional extended tag, corresponding to the source of the put/prescribe request. The `step-done` command now includes two counts, representing the number of puts and prescribes performed by the given step, respectively. Finally, the only attribute of a step that's relevant in the checkpoint is if the step is `done`, therefore we restrict the *Attributes* sort to only that value.

It should be noted that the checkpoint model does not include an *OutCommand* sort. This is because the checkpoint should only *react* to changes in the graph state, not influence it. Furthermore, we assume that restarting is a separate process that reads the entire state of a checkpoint rather than interacting with it.

5.1.3 Rewrite Rules

In this section, we describe the rewrite rules for the CnC checkpoint model.

CONFIGURATION:

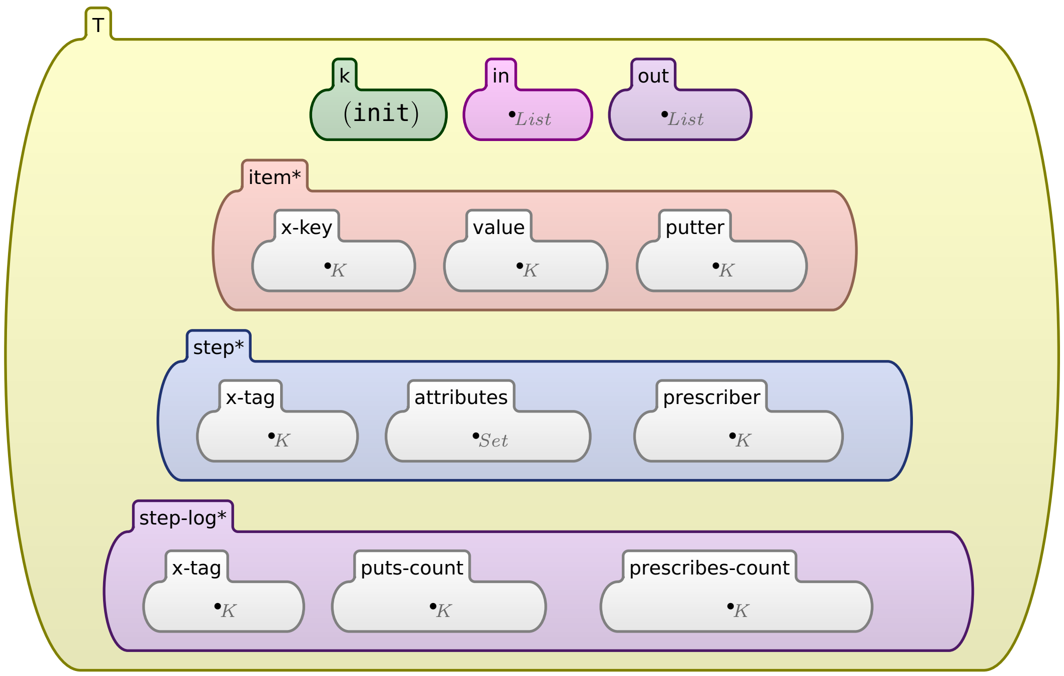


Figure 5.1 : K configuration for unoptimized CnC checkpointing.

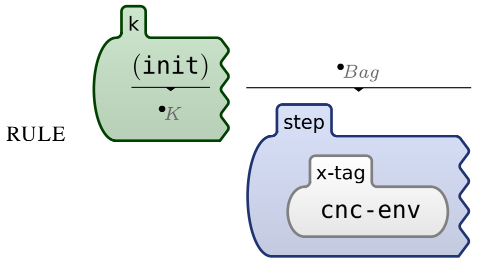
SYNTAX *EnvXTag* ::= `cnc-env`

SYNTAX *DepPair* ::= *EnvXTag*
| *IDExp TagExp* [seqstrict]

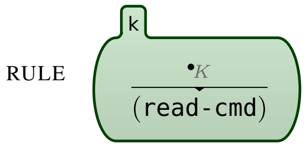
SYNTAX *InCommand* ::= (`init`)
| (`put` *DepExp DepExp DataExp*) [seqstrict]
| (`prescribe` *DepExp DepExp*) [seqstrict]
| (`step-done` *DepExp DataExp DataExp*) [seqstrict]
| *Halt*

SYNTAX *Attributes* ::= `done`

Figure 5.2 : Syntax changes for the checkpoint model.



(a) Initialize the environment tracker



(b) Read a new command from input

Rule Set 5.1: Rewrite rules for initializing the checkpoint and reading in commands.

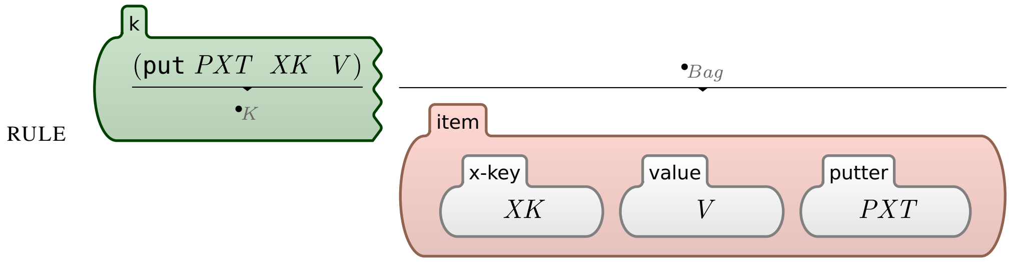
Initialization

Since the ⟨*k*⟩ cell is initialized to the value `(init)` (see figure 5.1), rule 5.1a is the first rule to be applied in any checkpoint model execution. This rule simply adds the `cnc-env` step to the checkpoint. If this step does not have the `done` attribute, then the environment may not have finished initializing the CnC graph. In that case we must assume the checkpoint is invalid, as described in section 4.3. Rule 5.1b reads a new command from *stdin* whenever the computation cell becomes empty.

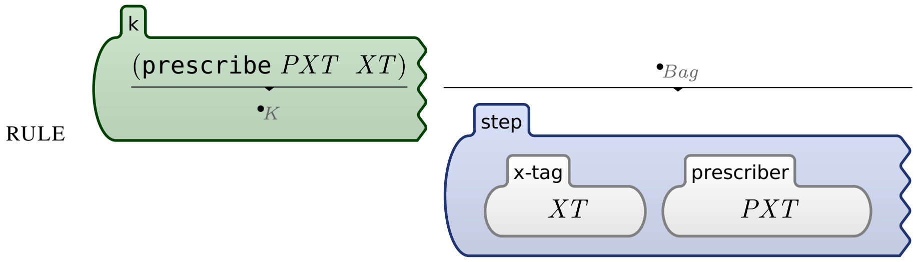
Input Commands

Rule set 5.2 includes all of the commands used to add new information to the checkpoint state. The `put` and `prescribe` commands—shown in rules 5.2a and 5.2b—each include the value PXT, which is the *putter's* or *prescriber's extended tag*. These values are used to verify that all of the puts and prescriptions made by a given step have been recorded in the checkpoint before marking a step with the `done` attribute. The `step-done` command in rule 5.2c contains the total number of puts and prescribes for a given step, which allows us to verify that all are present by a simply counting.

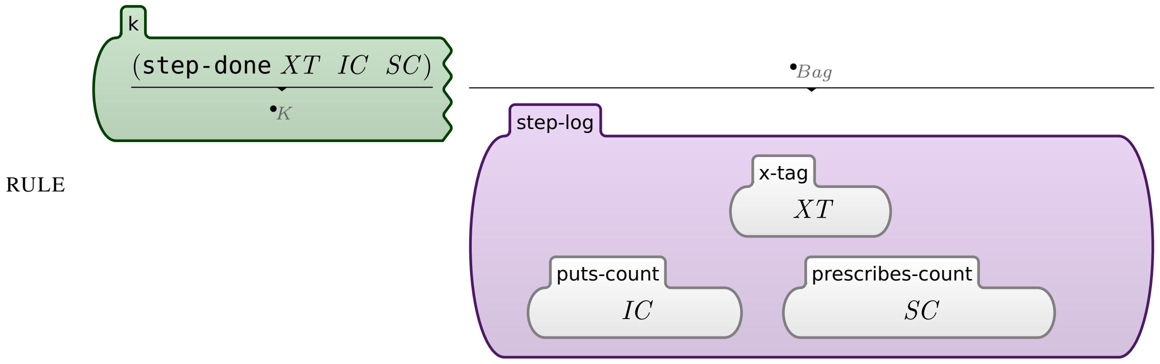
Note that a `get` command is not included in this model. Since item-gets are summarized by the ⟨*get-count*⟩ included in each ⟨*step-log*⟩, no other command is



(a) Put an item

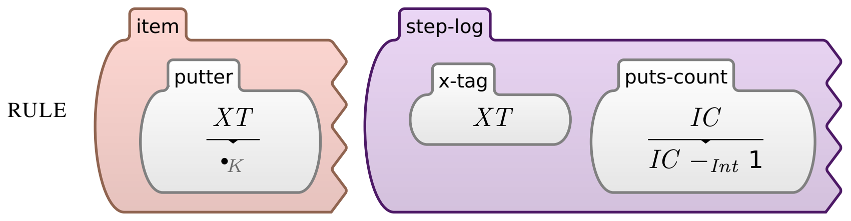


(b) Prescribe a step

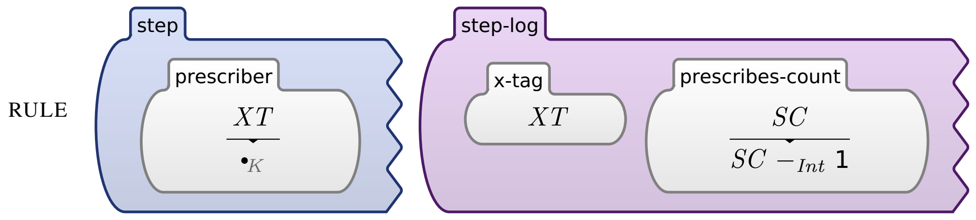


(c) Provide a summary of a step's execution

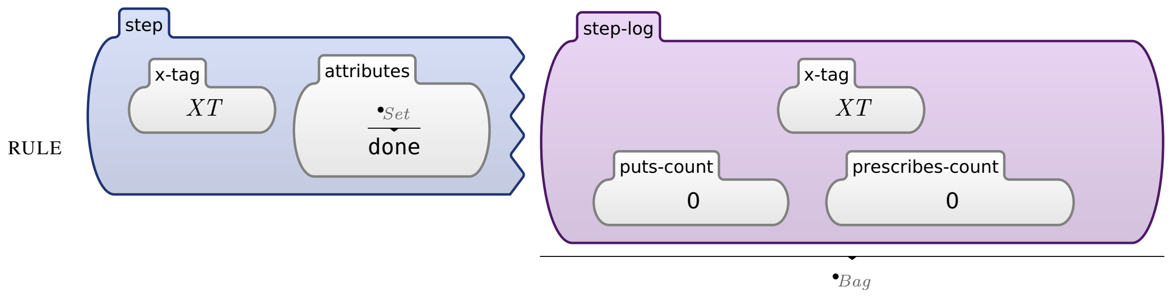
Rule Set 5.2: Rewrite rules for adding item and step information to the checkpoint.



(a) Accounting for a put



(b) Accounting for a prescription



(c) Step is done

Rule Set 5.3: Rewrite rules to account for a step's outputs, and then mark it as done.

needed to track them.

Step Accounting

As alluded to in section 5.1.3, the checkpoint must derive the fact that a step is `done` independently of the running graph. Since we assume all messages between the running CnC application and the checkpointing process are asynchronous (see section 3.1.1), we cannot infer that all of the outputs of a given step are reflected in the checkpoint simply based on the presence of a `step-done` command. Instead, we use

⟨*step-log*⟩ cells to make an independent accounting of each step before we can assume that it is `done` and therefore would not need to be rerun in the event of a restart. We account for all the outputs of a given step by decrementing the ⟨*puts-count*⟩ for each ⟨*putter*⟩ matching the extended tag of the corresponding ⟨*step-log*⟩ (rule 5.3a), and do the same for the ⟨*prescribes-count*⟩ and each matching ⟨*prescriber*⟩. Only when both the ⟨*puts-count*⟩ and ⟨*prescribes-count*⟩ of a ⟨*step-log*⟩ have reached zero can we add the `done` attribute to the corresponding ⟨*step*⟩.

Note that we could also guarantee that all output is present before a step is marked as `done` by moving all the information included in the `put` and `prescribe` commands into ⟨*step-done*⟩. We chose not take this approach for two reasons. First, it would potentially delay when information is available in the checkpoint, limiting the amount of information that can be immediately restored upon a restart. Second, waiting to send all the outputs of a step at once would result in *bursty* rather than *continuous* I/O for the checkpoint, which may be very undesirable in some cases. For example, if a step produces hundreds of data items—each containing several megabytes of data—over a relatively long period of time, sending all of that data in one large burst would result in a much higher latency than would sending each item as it is produced.

5.2 The Restart Algorithm

Algorithm 5.1 outlines the process for performing a restart with a CnC checkpoint as input. As explained in theorems 4.3 and 4.4, we have no guarantee that the checkpoint has enough information to successfully restart the graph computation until the environment has finished its initial *puts* and *prescribes.* This is signaled in the checkpoint by a `step-done` command with `cnc-env` as the *x-tag.* Therefore, if the `cnc-env` step in the checkpoint is not marked with the `done` attribute, then we

```
Data: contents of a CnC checkpoint, including the cncEnvStep entry
Result: CnC graph state restored from checkpoint
1  if $done \notin cncEnvStep.attributes$ then
2      throw error: cannot restart from incomplete checkpoint
3  else
4      foreach entry in checkpoint do
5          if entry is an item then
6              put given item with key and value to collection
7          else if $entry\ is\ a\ step \wedge done \notin step.attributes$ then
8              prescribe given step with tag in collection
9          else
10             discard entry
```

Algorithm 5.1: CnC restart algorithm

cannot restart from the checkpoint. This is reflected in the condition on line 1 and the error on line 2.

If the environment completed initialization, then the process continues by handling each entry in the checkpoint in the loop on line 4. All *item* entries are restored (lines 5 and 6), as are all non-done steps (lines 7 and 8). Any other entry—i.e., a done step or a step-log—is ignored (lines 9 and 10).

Given that a step is never marked with the `done` attribute until all of its outputs are present in the checkpoint (see section 5.1.3), we can infer that all steps that were running will be re-run, either directly via restoration or indirectly by a restored ancestor. Furthermore, steps that are not restored need not be run since their outputs will be restored (excluding output steps which were also done). Finally, any step that ran partially or completely but was restored during restart cannot corrupt the graph state since the steps are idempotent (theorem 4.1).

It follows from the logic above that all needed data items are also restored. All outputs of a step must be present in the checkpoint before it is considered done, and all non-done steps will be re-run in the restored graph; therefore, any item missing from the checkpoint must be reproduced by its parent step.

The restored graph is equivalent to the original graph, and CnC graphs are deterministic (theorem 4.5); therefore, the final result of the restored graph must be identical a complete run of the original graph.

5.2.1 Observations

An interesting property of this algorithm is that computation can begin while the restart is still in progress. The restart process is just a series of puts and prescribes, which is the same as the environment initializing the graph state in any CnC application. The only distinction is that, in the case of a restart, the initial graph data is being read from a checkpoint rather than being generated by the application environment.

5.3 Example: Restart with Matrix Multiplication

Here we run through a very simple example of the checkpoint/restart process for the sample application described in section 3.2.2.

As we can see by reading the graph initialization function (lines 63–80 of listing 3.2), the environment puts all the entries for `coll-a` and `coll-b` (the input matrices). In addition, the environment statically prescribes all the steps for the entire computation (notice that the `prescribe-step` function does not appear in either of the step function declarations). Therefore, if any of those step or item instances are not present in the checkpoint, then the *cnc-env* step will not be done, and the checkpoint must be discarded. However, as long as those instances are present, then *cnc-env* is done and the checkpoint is a valid source for a restart.

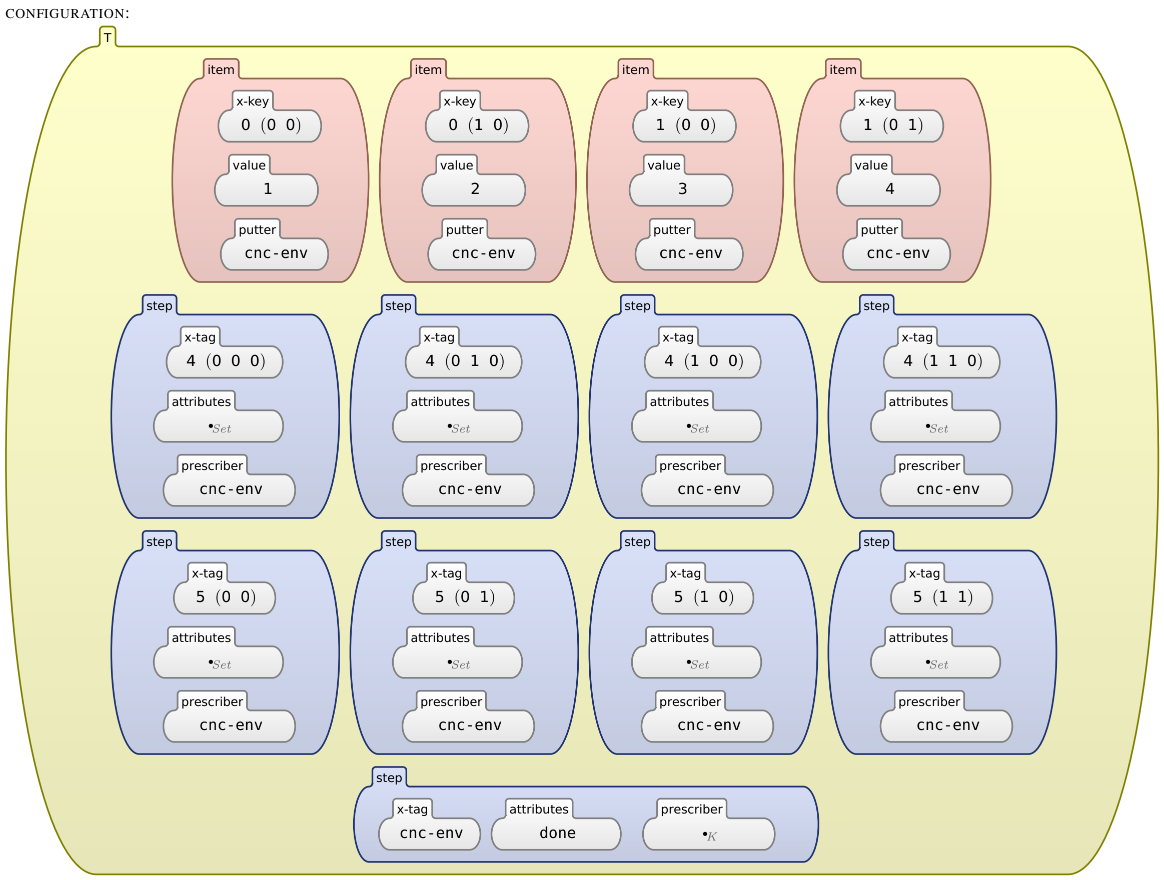


Figure 5.3 : State of the CnC checkpoint for the matrix multiplication application after the environment completes its puts and prescribes. Recall from listing 3.2 that the integers 0, 1, 4 and 5 are the unique IDs for the collections *coll-a*, *coll-b*, *mult-step* and *sum-step*, respectively.

mult-step	sum-step	coll-a	coll-b
$\langle 0,0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle \to 1$	$\langle 0,0\rangle \to 3$
$\langle 0,1,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle \to 2$	$\langle 0,1\rangle \to 4$
$\langle 1,0,0\rangle$	$\langle 1,0\rangle$		
$\langle 1,1,0\rangle$	$\langle 1,1\rangle$		

Table 5.1 : Step and item instances added to the graph state by the environment in the matrix multiplication example from section 3.2.2. Each column represent a separate collection. Step instances are represented by their tags, and item instances are represented by their key-value pairs.

Let us use the simple multiplication example from section 3.2.2:

$$\begin{bmatrix}1\\2\end{bmatrix} \times \begin{bmatrix}3 & 4\end{bmatrix}$$

In this case, the graph initialization function will create the step and item instances outlined in table 5.1. For this example we assume that all of these item and step instances were successfully put or prescribed by the environment, and that all of this information is reflected in the current checkpoint state, as illustrated in figure 5.3.

Assume that $(\!|$ mult-step: 0,0,0 $|\!)$ completes, multiplying $[\![$ coll-a: 0,0→1 $]\!]$ by $[\![$ coll-b: 0,0→3 $]\!]$ and producing $[\![$ coll-products: 0,0,0→3 $]\!]$. This would result in two update messages to the checkpoint: a *step-done* and a *put*.

If only the *step-done* message arrives, then $(\!|$ mult-step: 0,0,0 $|\!)$ is not yet considered done because it still has outstanding outputs, and the restorable checkpoint remains unchanged. If only the *put* arrives, then $[\![$ coll-products: 0,0,0→3 $]\!]$ is added to the checkpoint. In the case of a restart, $[\![$ coll-products: 0,0,0→3 $]\!]$ would be restored to the graph during initialization; however, $(\!|$ mult-step: 0,0,0 $|\!)$ was not marked as *done* in the checkpoint, meaning that it will be re-run along with all of the other step instances listed in table 5.1, resulting in a duplicate *put* of $[\![$ coll-products: 0,0,0 $]\!]$.

If both the *step-done* and *put* arrive, then ⟦ coll-products: 0,0,0→3 ⟧ is added to the checkpoint, and ⦇ mult-step: 0,0,0 ⦈ will be marked *done* since all of its outputs are present in the checkpoint. In the case of a restart, ⟦ coll-products: 0,0,0→3 ⟧ would be restored; however, in this case ⦇ mult-step: 0,0,0 ⦈ was marked *done*, and therefore that step instance not re-run, and it does not produce a duplicate item instance.

5.4 Example: Restart with Pascal's Triangle

Now we run through a simple example of the checkpoint/restart process for the sample application described in figure 2.3 and section 3.2.1. The source code for this example is shown in listing 3.1. For this example we will use the computation of ${}_2C_1$, as described in section 2.1.2 and illustrated in figure 2.5.

Assume that all *put*, *prescribe* and *step-done* updates from ⦇ edge-step: 0,0 ⦈ and ⦇ edge-step: 1,1 ⦈ arrived in the checkpoint. However, after ⦇ edge-step: 1,0 ⦈ finishes producing its outputs the step crashes, causing that step's *put* and *prescribe* updates to arrive in the checkpoint, but not the *step-done* update. None of the step instances from row 2 have produced any outputs. In this case, ⦇ edge-step: 0,0 ⦈ and ⦇ edge-step: 1,1 ⦈ are both *done* since all their outputs are accounted for. However, since the *step-done* message never arrived for ⦇ edge-step: 1,0 ⦈, the checkpoint cannot account for its outputs; therefore, it is not considered *done* even though both of its outputs are actually present in the checkpoint, as shown in figure 5.4.

If we did a restart from this checkpoint, we would restore the step and item instances listed in table 5.2. Since ⦇ edge-step: 1,0 ⦈ was restored along with both of its outputs, all *descendants* of that step (i.e., any node reachable from the corresponding node via the directed edges in figure 2.5) will be duplicated in the restarted execution. In other words, there will be two copies of every instance ⦇ edge-step: i,0 ⦈ and ⟦ pascal-

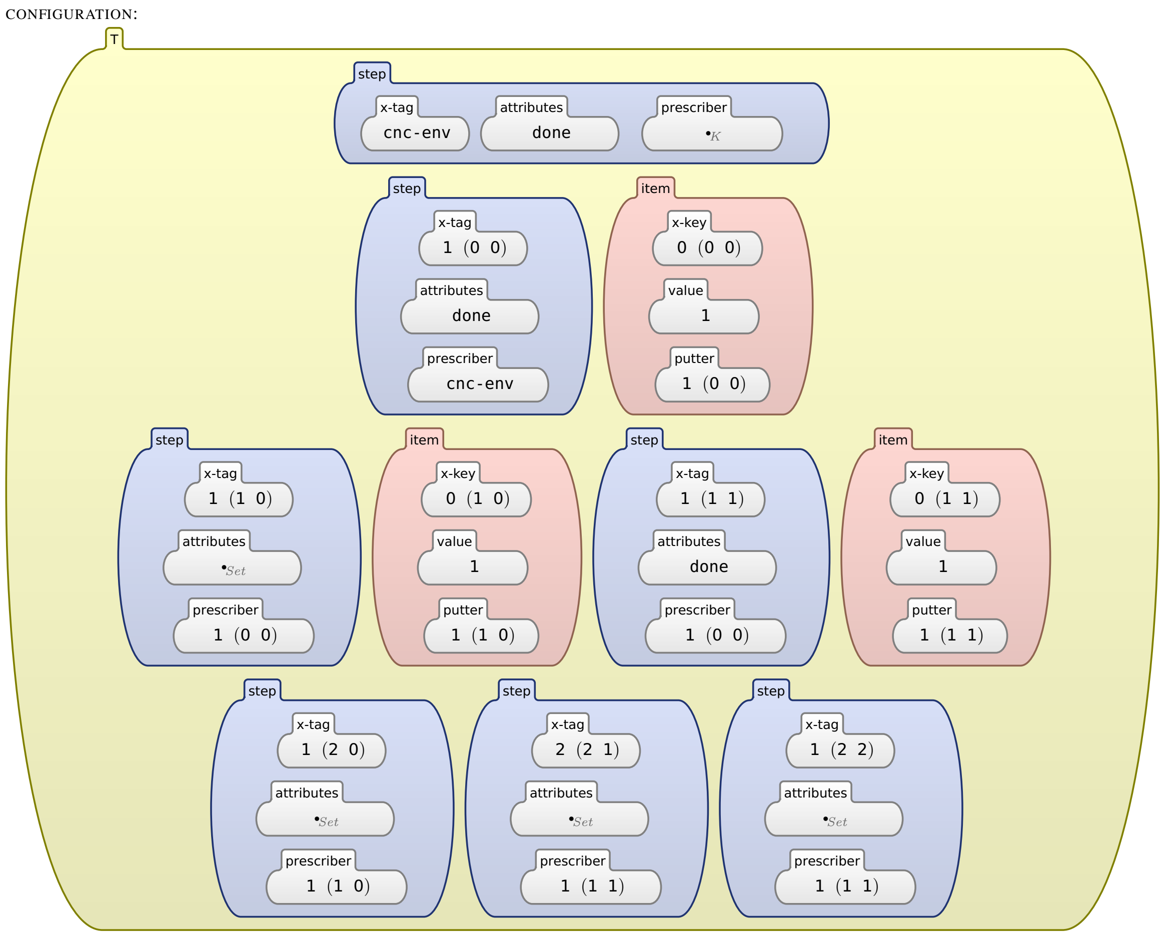


Figure 5.4 : State of the CnC checkpoint for the Pascal's Triangle application execution scenario described in section 5.4. Recall from listing 3.1 that the integers 0, 1, and 2 are the unique IDs for the collections *pascal-entries*, *edge-step* and *inner-step*, respectively.

edge-step	inner-step	pascal-entries
$\langle 1,0\rangle$	$\langle 2,1\rangle$	$\langle 0,0\rangle \to 1$
$\langle 2,0\rangle$		$\langle 0,1\rangle \to 1$
$\langle 2,2\rangle$		$\langle 1,1\rangle \to 1$

Table 5.2 : Step and item instances added to the graph state by the environment during a sample restart of the ${}_2C_1$ calculation described in this section.

entries: i,0 ⟧ for $1 \leq i \leq n$. In this case, ⦇ edge-step: 1,0 ⦈ only has two descendants—⦇ edge-step: 2,0 ⦈ and ⟦ pascal-entries: 2,0 ⟧—because $n = 2$. However, performing the same restart for a computation with a larger value of n would result in $n-1$ repeated step computations and $n-1$ repeated puts of the corresponding item instances.

5.5 Summary

We have described a model for computing checkpoints for a CnC graph execution based on asynchronous update messages from the CnC runtime. We have shown that, based on the properties from section 4.3, these checkpoints always produce a valid checkpoint for use with the restart algorithm described in algorithm 5.1. Although the result of a restart is guaranteed to produce the correct result, we saw by our example in section 5.4 that naively applying this algorithm can result in large amounts of duplicate computation. In chapters 6 and 7 we modify our existing models to address this and other inefficiencies.

Chapter 6

Execution Frontiers in CnC

In theorem 4.2 we proved that the state of a CnC graph grows monotonically. However, this also implies that the memory requirements of a CnC application must grow monotonically throughout the execution, which is not practical in general. In this chapter we introduce the concept of a CnC *execution frontier* (XF), which will allow us to optimize the memory footprint of a CnC application by removing unneeded data. We show that this optimized version of the CnC runtime is functionally equivalent to the unoptimized version defined in chapter 4.

6.1 The CnC Execution Frontier

The *execution frontier* of a CnC execution graph is defined as the set of all live step and item instances in the graph; furthermore, an instance is *live* if and only if the attribute set is neither empty nor contains an *end-of-life* attribute [23]. For CnC steps, the *done* attributes indicates end-of-life. To indicate end-of-life for CnC items, we introduce a new attribute: *dead*. An item is considered alive as long as a step or the environment will still *get* the item's value at some point in the future. After the item's value is read for the last time, it is then marked with the *dead* attribute.

The execution frontier—or set of live instances—can be thought of as *flowing* through the set of all possible step and item instances in a CnC graph. Figure 6.1 illustrates an example of the execution frontier in a simple 3-point stencil application

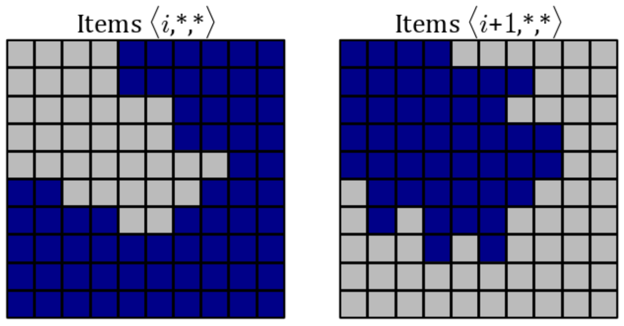


Figure 6.1 : A possible snapshot of the state of the data items for a 3-point (Γ-shaped) stencil computation on a matrix. Each item is represented by a 3-tuple: $\langle iteration, row, column\rangle$. Dark blue cells represent the *live* data. Light gray cells in iteration i are *dead*, whereas the gray cells in iteration $i+1$ have not yet been put. Note that the pattern in the two iterations is not symmetric.

as it *flows* through an item collection. The two grids represent the matrices for two consecutive iterations of the stencil computation. The dark blue cells in the grids represent the items (corresponding to individual matrix entries) that are *live* at the time of the snapshot. Note that the set of light gray (dead) cells with coordinates in the grid for iteration i corresponds exactly to the following equation:

$$dead(i,x,y) \Leftrightarrow live(i+1,x,y) \wedge live(i+1,x+1,y) \wedge live(i+1,x,y+1) \qquad (6.1)$$

In other words, an entry E in iteration i becomes *dead* once all three items from iteration $i+1$ that depend on E in the Γ-shaped stencil are *live*. This follows from our rule for marking an item with the *dead* attribute, since the entry E will not be read again after all its dependent entries in the next iteration have been computed.

6.1.1 The Leading Edge

As the execution frontier flows through the graph state-space, we say that new step and item instances enter through its *leading edge*. This leading edge is realized in every CnC implementation—including our models from chapters 4 and 5—by the *prescribe* and *put* operations, which add new instances to the step and item collections. An implementation of CnC that did not support a leading edge could only represent the identity function since it could neither run any step computations nor produce any new data items.

6.1.2 The Trailing Edge

As the execution frontier flows through the graph state-space, we say that step and item instances that have reached end-of-life exit through its *trailing edge*. The trailing edge of the step collections is fairly trivial to track in the model from chapter 4. A step instance leaves through the trailing edge once it acquires the *done* attribute. However, tracking the trailing edge through the item collections is not quite as straightforward due to the difficulties of determining when the last use of an item instance takes place [8]. Consequently, not all CnC implementations realize the trailing edge of the execution frontier.

Several methods have been proposed for calculating the liveness of item instances. One option is to provide a *get count* (or *reference count*) function. This can be a function explicitly declared by the user, or might be generated indirectly as described by Budimlić et al. with slicing annotations [8]. Another possibility is to carefully reuse memory via a relationship described by *folding function* [7].

In this work we choose the get-count function as our method for tracking the trailing edge through item collections. We made this choice both based on the simplicity

```
43 ;; Get count function for the pascal-entries collection
44 (defn entry-get-count [ctx [row col]]
45   (let [n (get-static ctx :n)
46         k (get-static ctx :k)]
47     (cond
48       ; All entries in the last row are never read,
49       ; except the kth entry, which is read once as output
50       (== row n) (if (== col k) 1 0)
51       ; Row 0 is not read
52       (== row 0) 0
53       ; Edge entries are only read once
54       (or (== k 0) (== k n)) 1
55       ; Other entries are read twice
56       :else 2)))
```

Listing 6.1: The get count function for the *pascal-entries* item collection for the Pascal's Triangle CnC application. This is a snippet from the full program code in listing 3.1, beginning at line 43.

```
57        :itemcolls {coll-a        (constantly b-cols)
58                    coll-b        (constantly a-rows)
59                    coll-c        (constantly 1)
60                    coll-products (constantly 1)}
```

Listing 6.2: Get count functions for the matrix multiply application, given within its CnC graph specification. This is a snippet from the full program code in listing 3.2, beginning at line 57.

of using counters, and on the fact that get counts (also known as use counts) are the only trailing-edge strategy incorporated in a production implementation of CnC [5].

An example of an explicit get-count function is provided on line 43 of listing 3.1, which is replicated in listing 6.1 for the reader's convenience. While building n rows of Pascal's Triangle, most entries in the triangle are used twice: once to calculate the item directly below, and again to calculate the item below and one column to the right. This is why `entry-get-count` returns 2 in the default case. Entries on the left and right edges of the triangle are used once. In the last row of the triangle, the k^{th} column is used once by the environment to get the result of ${}_nC_k$, while none of the other columns' entries are used at all; hence, `entry-get-count` returns 1 for column k of row n, and 0 for the other entries in that row.

Since the matrix multiplication example from listing 3.2 includes four item collections, it also needs four get count functions. These functions are given inline with the CnC graph specification on lines 57–60, which is replicated in listing 6.2 for the reader's convenience. Clojure's `constantly` function returns a new constant function that ignores any arguments and always returns the same value. Each element in matrix `a` is used once for each column in `b`, and each element in `b` is used once for each row in `a`. Hence, the get functions for `coll-a` and `coll-b` are `(constantly b-cols)` and `(constantly a-rows)`, respectively. Since the intermediate products and the entries in the output matrix are each only used once, both `coll-c` and `coll-products` have the get function `(constantly 1)`.

6.1.3 Observations

Checkpoints are a type of execution frontier, in that they capture a snapshot of the live state (and possibly additional data) of the executing graph. The execution frontier embodies the concepts of data- and control-encapsulation as covered in theorems 4.3 and 4.4.

6.2 Model

We now introduce an update to the model from chapter 4, realizing the trailing edge of the execution frontier in both item and step collections.

6.2.1 Configuration

Figure 6.2 shows the updated configuration for a CnC runtime with a trailing edge. The only changes made to the configuration compared to the configuration described in section 4.2.1 are the additions of ‹*get-count*› and ‹*attributes*› within ‹*item**›.

6.2.2 Syntax

The changes to the syntax are also minor, and are outlined in figure 6.3. An additional argument was added to the `put` corresponding to the item's get count. All other *InCommand* productions remain unchanged. We also inserted a production to declare the newly added `dead` attribute for items.

6.2.3 Rewrite Rules

Rule sets 6.1 and 6.2 summarize the changes to the rewrite rules from section 4.2.3. Rule set 6.1 are modifications to the existing `put` and `get` commands to handle the new ‹*get-count*› cell. When an item is *put*, the total get-count is included with the new item instance. Each time the user *gets* an item's ‹*value*›, the corresponding ‹*get-count*› is decremented. Rule set 6.2 are the new additions for allowing instances to flow out the trailing edge of the execution frontier. First, when an item's ‹*get-count*›

CONFIGURATION:

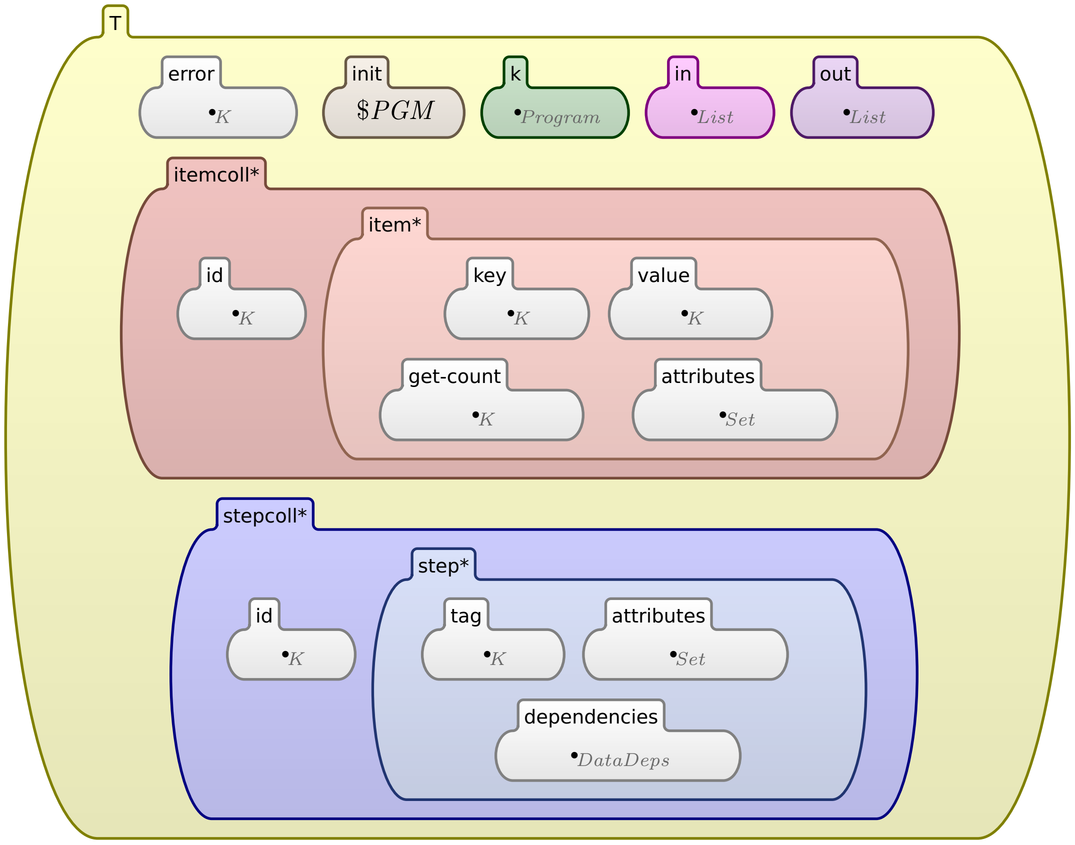


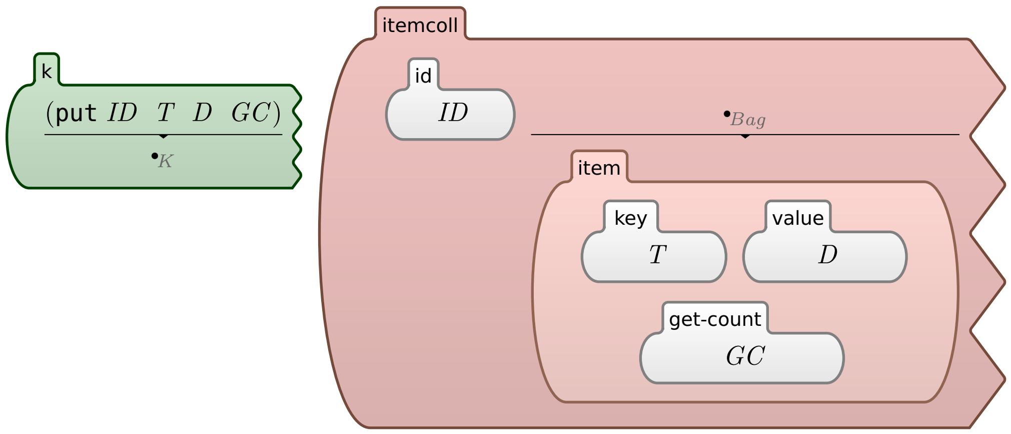
Figure 6.2 : K configuration for a CnC runtime with a trailing edge.

SYNTAX *InCommand* ::= (`put` *IDExp* *TagExp* *DataExp* *DataExp*) [seqstrict]

SYNTAX *ItemAttr* ::= `dead`

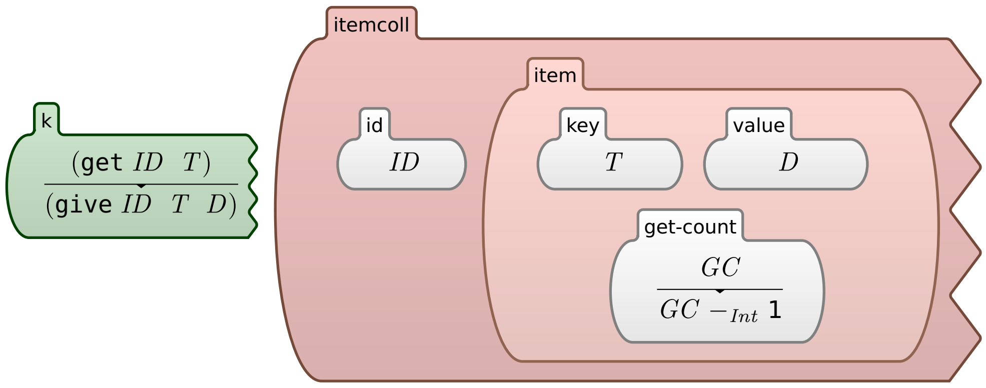
Figure 6.3 : Summary of syntax changes for accommodating a trailing edge.

RULE



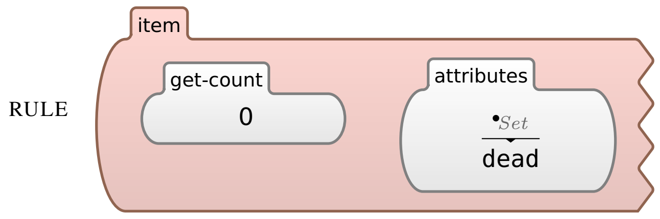
(a) Item put

RULE

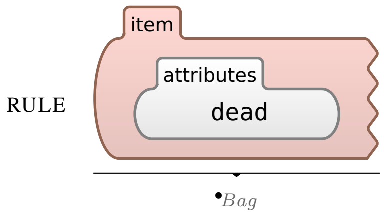


(b) Item get

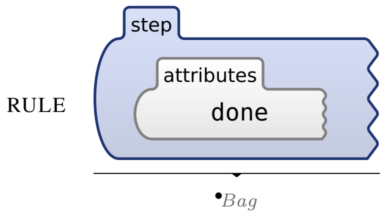
Rule Set 6.1: Updated *put* and *get* commands, tracking an item's *get-count*.



(a) Item is dead



(b) Remove dead item



(c) Remove done step

Rule Set 6.2: New rewrite rules for realizing the trailing edge.

reaches zero, the `dead` attribute is added, as shown in rule 6.2a. Next, as shown in rules 6.2b and 6.2c, an instance is deleted when it acquires a `dead` or `done` attribute.

6.3 Changes in Key Properties

Maintaining a get count and removing items based on that get count affects some of the key properties of CnC that we defined in section 4.3. Specifically, we can no longer consider steps to be idempotent since re-running a step would cause an extra decrement on the get count of each step input. This is a direct result of our definition of an item's end-of-life; we could never decide when an item becomes dead if a step that accessed that item could be re-run at an arbitrary point in the future. However, given the added assumption that no steps are ever re-run, the other properties still hold. Therefore, in the updated model it is considered a programmer error if a single step instance is ever prescribed twice in the same graph execution.

Theorem 6.1

Given that no step is ever re-run, the updated model is equivalent to the original model for all error-free programs.

Proof 6.1 An item is given the *dead* attribute only when all get operations for that item have been processed. If there are no more get operations for the item, then its presence or absence in the item collection cannot be observed. Therefore, a *dead* item can be safely removed from an item collection without influencing the behavior of the graph. In other words, since it is not possible to observe the absence of the removed items, the two models are functionally equivalent. ■

Theorem 6.1 proves the functional equivalence of our updated model and the original model from chapter 4 at the cost of the idempotent step property from theorem 4.1. Since the restart algorithm described in section 5.2 hinges upon the ability to re-run any previously partially-run or fully-run step instance, algorithm 5.1 is not valid under our updated model.

6.4 Summary

We have realized the full execution frontier—including both the *leading* and *trailing* edges—in a model built on that introduced in chapter 4. We did this with minimal changes to the syntax and configuration, while only modifying two rewrite rules and only adding three new rewrite rules. We then proved the functional equivalence of this updated model with the original model, obviating the requirement that the CnC runtime's memory footprint must grow monotonically throughout graph execution; however, by considering it an error to prescribe a single step instance twice in the same graph execution, we have removed one of the key properties of our original CnC model. Without the idempotent step property our naive checkpoint/restart system from chapter 5 cannot function correctly. We redefine our checkpoint/restart method to deal with this new restriction in chapter 7.

Chapter 7

Extended Model of CnC Checkpointing

In this chapter, we modify our model for tracking CnC checkpoints in order to avoid re-running previously completed steps. We add this restriction based on the effects of using get-counts on items to realize the trailing edge of the execution frontier in our item collections, as discussed in chapter 6. This requires us to carefully derive the current state of the execution frontier for our checkpoint based on the internal state of the checkpoint. The checkpointing process runs asynchronously of the graph execution, and therefore it must independently derive both the leading and trailing edges of the execution frontier in order to maintain a consistent state for a possible future restart.

7.1 The Modified Checkpoint Model

7.1.1 Configuration

Figure 7.1 illustrates the modified K configuration of our CnC checkpoint model. The configuration encapsulates the complete state of the model. Most of the cells should be familiar from the previously introduced configurations. We outline the functions of the newly added cells below.

⟨***staged***⟩ Holds all the data that has not yet entered the XF.

⟨***pre-item****⟩ Each represents an item that has not yet entered the XF.

⟨***pre-step****⟩ Each represents a step that has not yet entered the XF.

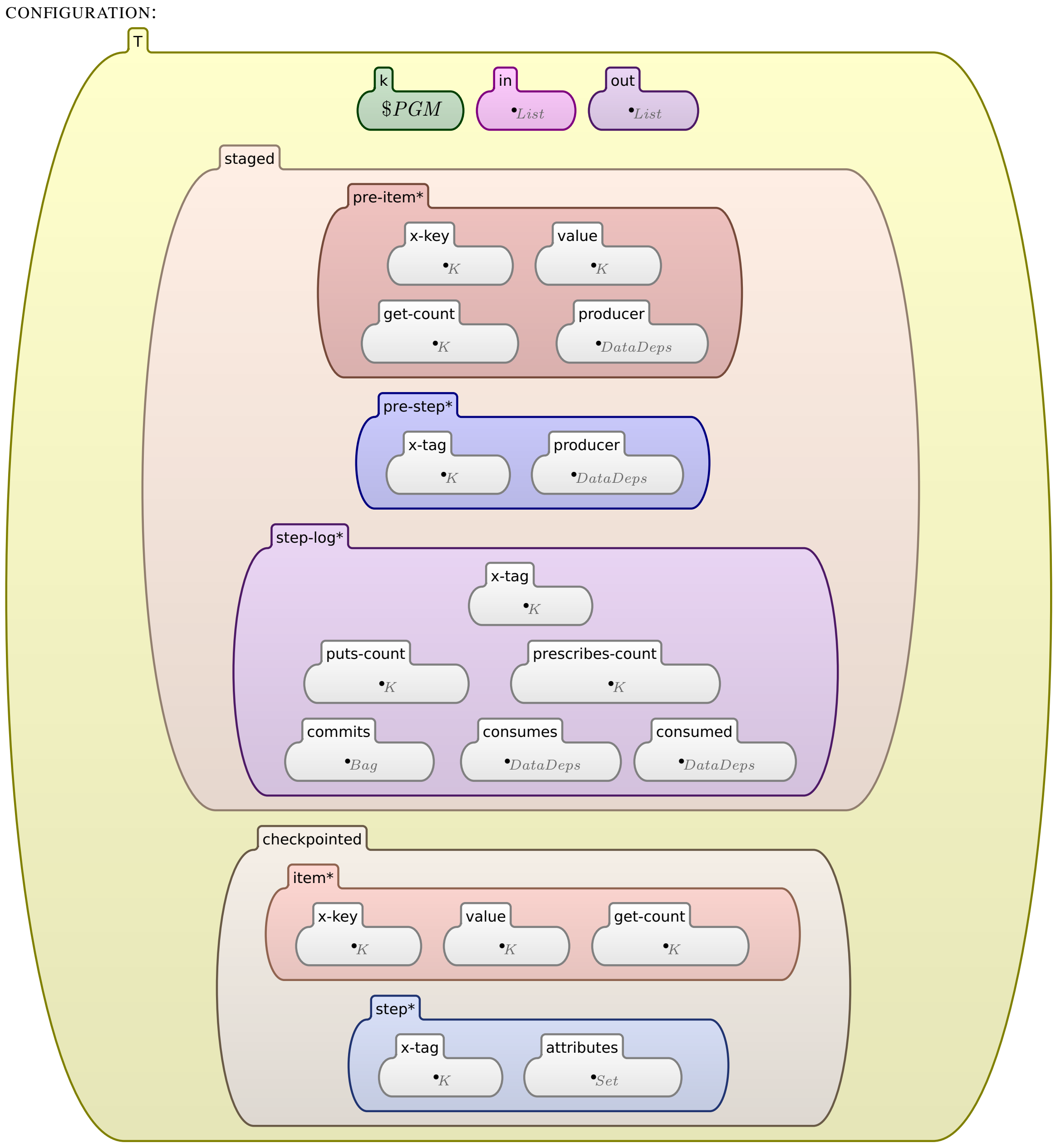
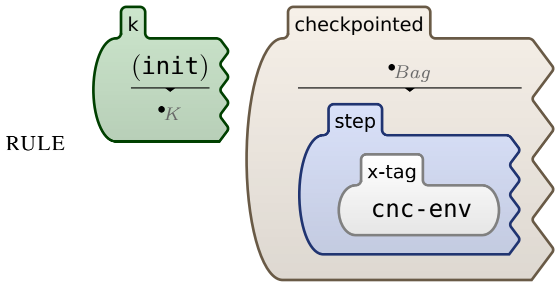
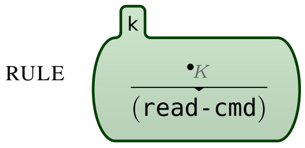


Figure 7.1 : K configuration for CnC checkpointing with the XF trailing edge.



(a) Initialize the environment tracker



(b) Read a new command from input

Rule Set 7.1: Rewrite rules for initializing the checkpoint and reading in commands.

⟨***checkpointed***⟩ Holds the current XF. A restart will only consider items and steps contained in this cell.

⟨***commits***⟩ Holds the ⟨*item*⟩ and ⟨*step*⟩ instances produced by this step, all of which will be added to ⟨*checkpointed*⟩ when this step is done.

⟨***consumes***⟩ Holds the set of items that the step consumes as inputs.

⟨***consumed***⟩ Holds entries from ⟨*consumes*⟩ that have been satisfied.

7.1.2 Syntax

All of the syntax necessary for the updated checkpointing model has been introduced in previous sections; therefore, we do not introduce any new syntax for this variant on the model.

7.1.3 Rewrite Rules

In this section, we describe the rewrite rules for the updated CnC checkpoint model. These rewrite rules are much more complex than those we specified for the previous models. This added complexity is a result of our requirement to independently derive the current flow of the execution frontier through the graph state.

Initialization

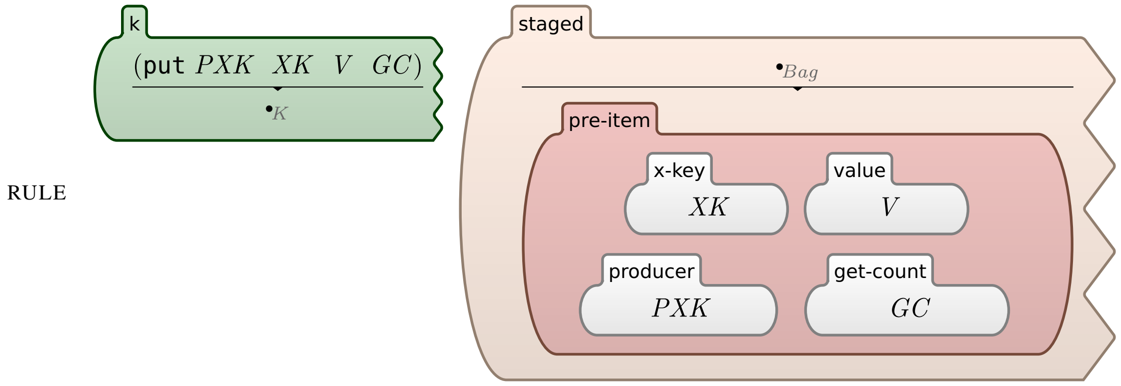
Rule set 7.1 describe the rewrite rules for initializing the updated checkpoint model. The initialization commands are mostly identical to those described in section 5.1.3. The only difference here is the presence of the ‹*checkpointed*› cell. The `cnc-env` step is put directly into the ‹*checkpointed*› cell because in the event of a restart we would always want to re-run the environment's graph initialization function if it was not yet done.

Input Commands

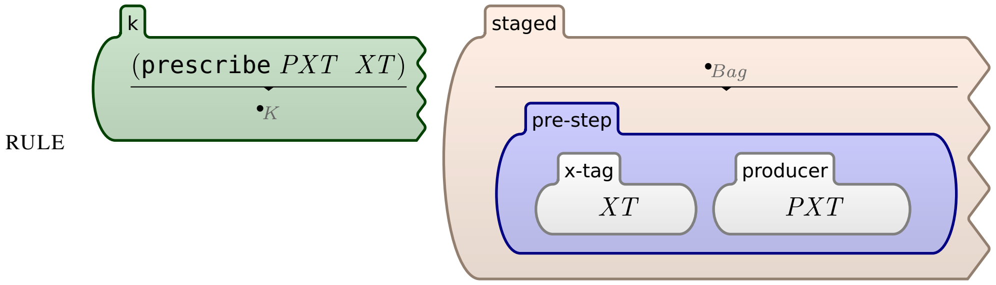
Rule set 7.2 shows the new handling of inputs to the checkpoint. Rather than directly *committing* step and item instances into the checkpoint, the instances are first *staged* by storing them in the ‹*staged*› cell. Although these instances were live in the graph execution, we cannot yet assume they have entered through the leading edge of the execution frontier in the checkpoint due to the lack of ordering guarantees with asynchronous communication between the two processes. The ‹*step-log*› cells are also stored in the ‹*staged*› cell since they are not collection instances and thus do not live in the execution frontier.

Step Accounting

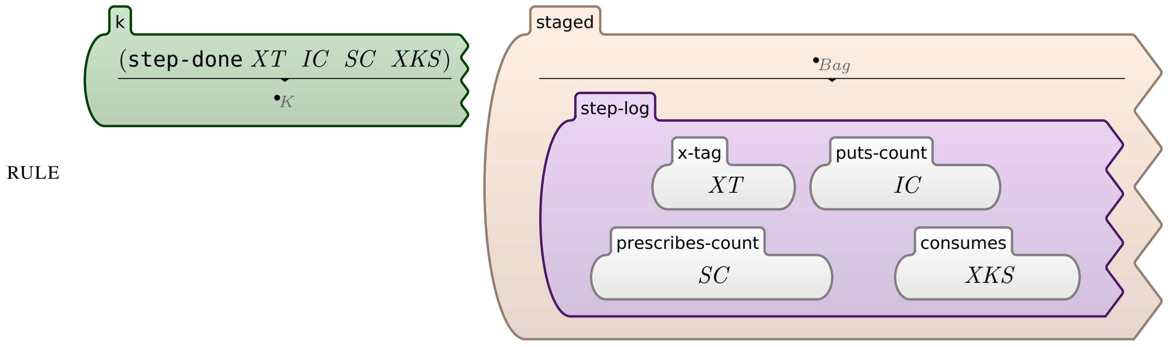
Rule set 7.3 shows how we track the outputs of a step in the checkpoint. As shown in Rule 7.3a, each ‹*pre-item*› is matched with the ‹*step-log*› of its producer. The contents of the ‹*pre-item*› are copied into the ‹*commits*› as an ‹*item*› so that when this step is complete all its outputs can be committed to ‹*checkpointed*›. Rule 7.3b similarly handles step prescriptions. In each case the step-log's corresponding *count* is decremented to track the number of remaining dependencies.



(a) Put an item



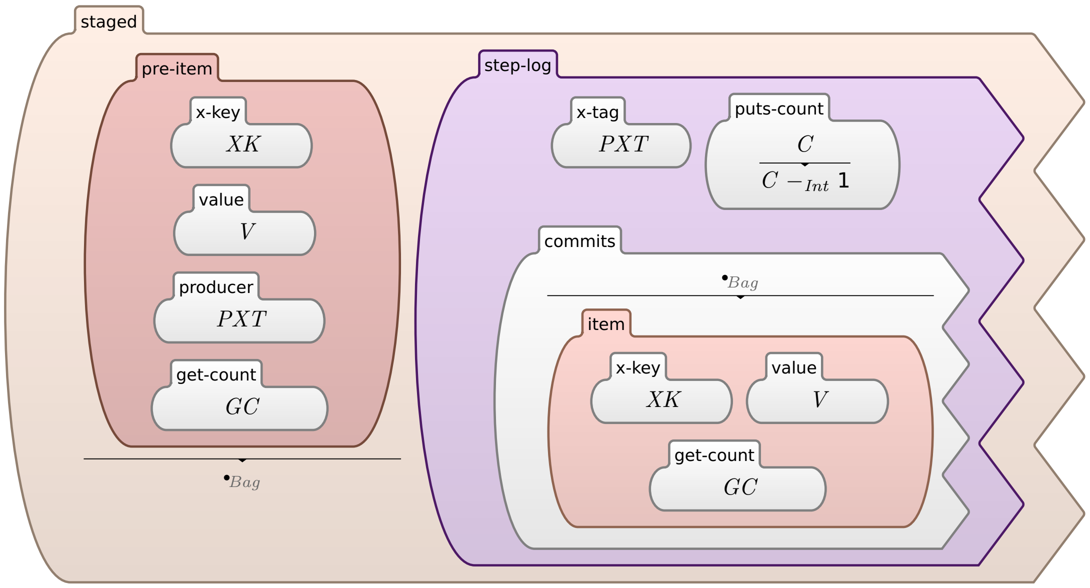
(b) Prescribe a step



(c) Provide a summary of a step's execution

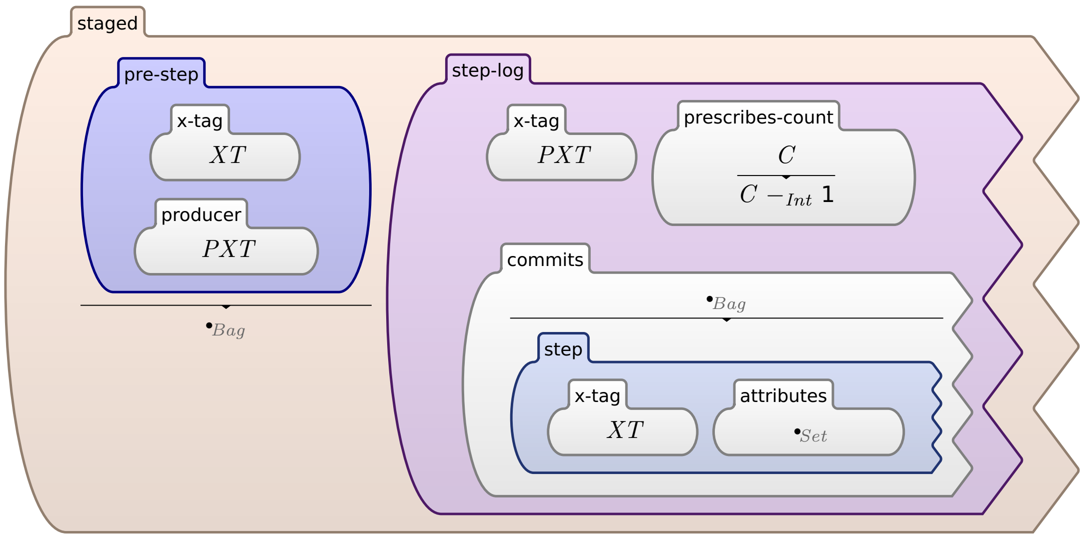
Rule Set 7.2: Rewrite rules for adding item and step information to the checkpoint.

RULE



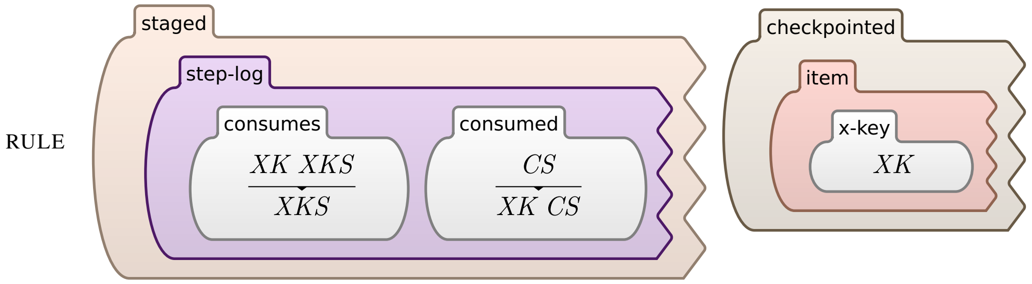
(a) Account for items put by a given step

RULE

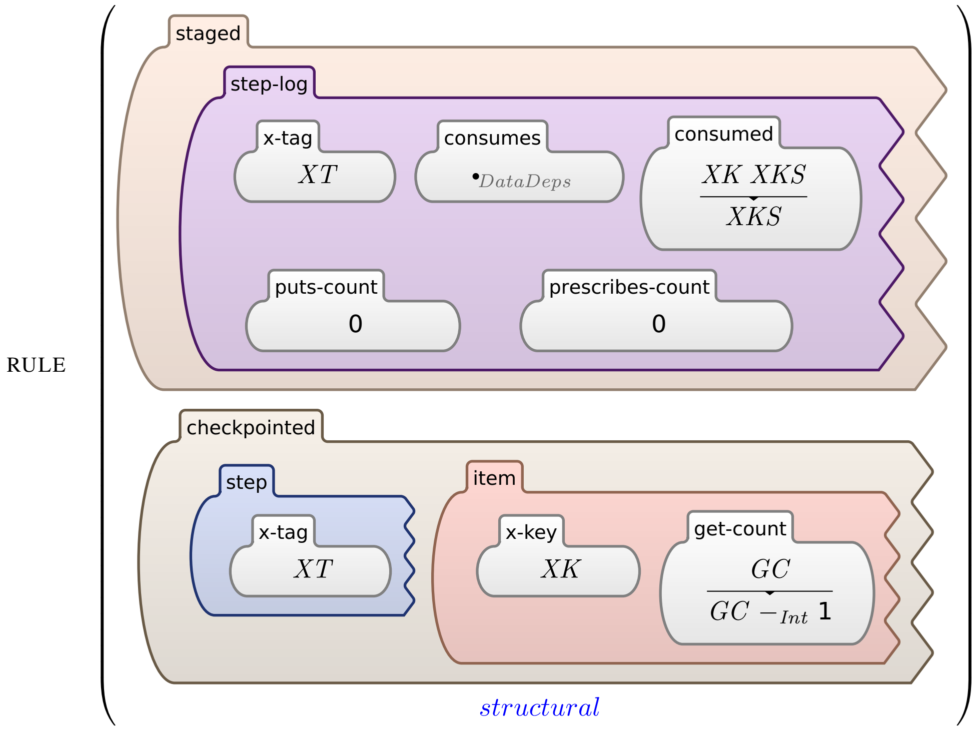


(b) Account for steps prescribed by a given step

Rule Set 7.3: Rewrite rules tracking step outputs.



(a) Satisfy a step's input dependence



(b) Decrement step input's get count

Rule Set 7.4: Rewrite rules for satisfying step input dependencies.

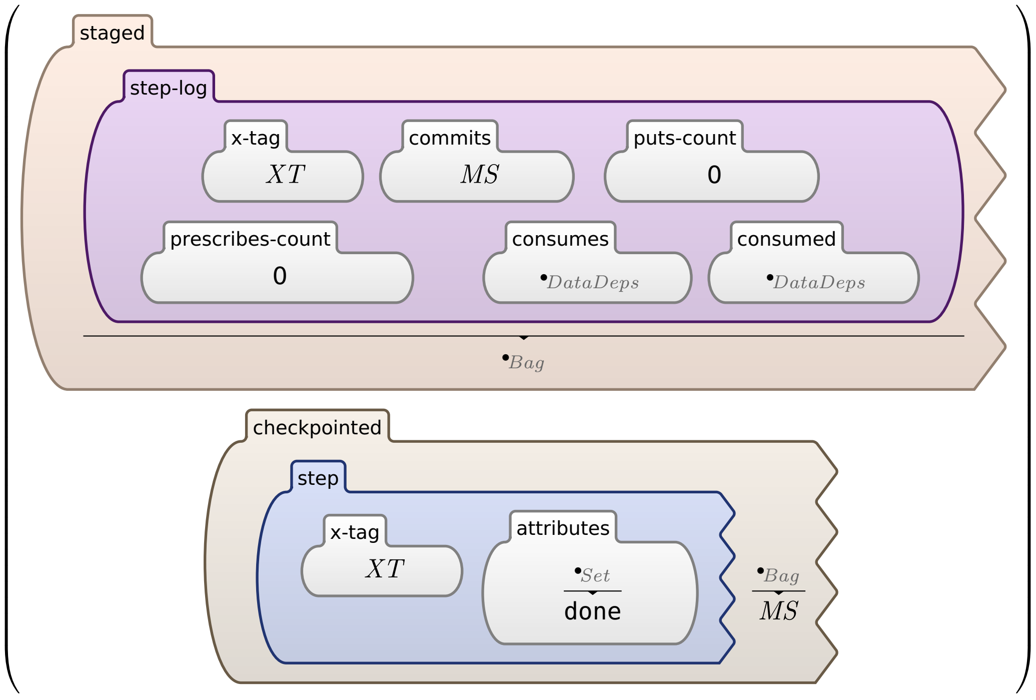
The inputs to a step must also be tracked, as shown in rule set 7.4. Before a step can run, all its inputs must first be live in the execution frontier. This means that each dependence listed in ⟨*consumes*⟩ must match with an ⟨*item*⟩ in ⟨*checkpointed*⟩ before the dependence can be moved to ⟨*consumed*⟩ (rule 7.4a). Rule 7.4b decrements the get count for each item indicated in ⟨*consumed*⟩ once all the step's dependencies have been satisfied. Rule 7.4b is marked as *structural* because these updates should all happen atomically at the step's completion.

Leading and Trailing Edge Flows

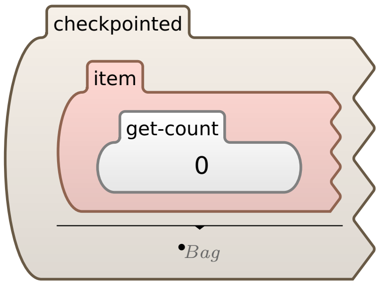
Rule set 7.5 accounts for all updates to the execution frontier in the checkpoint via the leading edge. When all of a step's input and output dependencies have been satisfied, the step is *done*, and all the instances in its ⟨*commits*⟩ cell are moved into ⟨*checkpointed*⟩. Aside from the `cnc-env` step (rule 7.1a), all step and item instances come into are added into the execution frontier by this rule. Rules 7.5b and 7.5c account for the trailing edge of the execution frontier, removing item and step instances once they have reached end-of-life.

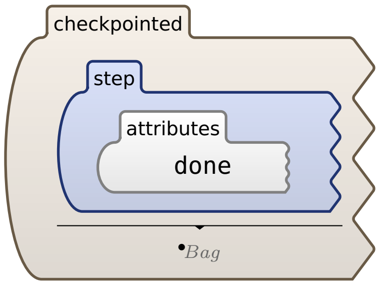
7.2 Restarting

The careful accounting in this checkpoint model's rewrite rules guarantee that step and item instances are included in the execution frontier (the ⟨*checked*⟩ cell) if and only if they are live based on the current state of the checkpoint. Steps and items enter the execution frontier only when their producer step is done, and they exit the execution frontier only when they have reached end-of-life. These two properties allow us to guarantee that a CnC graph restored from the execution frontier in our checkpoint will never re-run steps (since a step is only included if all its ancestors are

RULE 

(a) Commit all outputs of step when the step is done

RULE 

RULE 

(b) Remove dead item

(c) Remove done step

Rule Set 7.5: Rewrite rules for XF leading and trailing edges.

done), nor will it ever restore a copy of an item instance that will be re-put (because the item is not entered into the XF until its producer step and all ancestors are done).

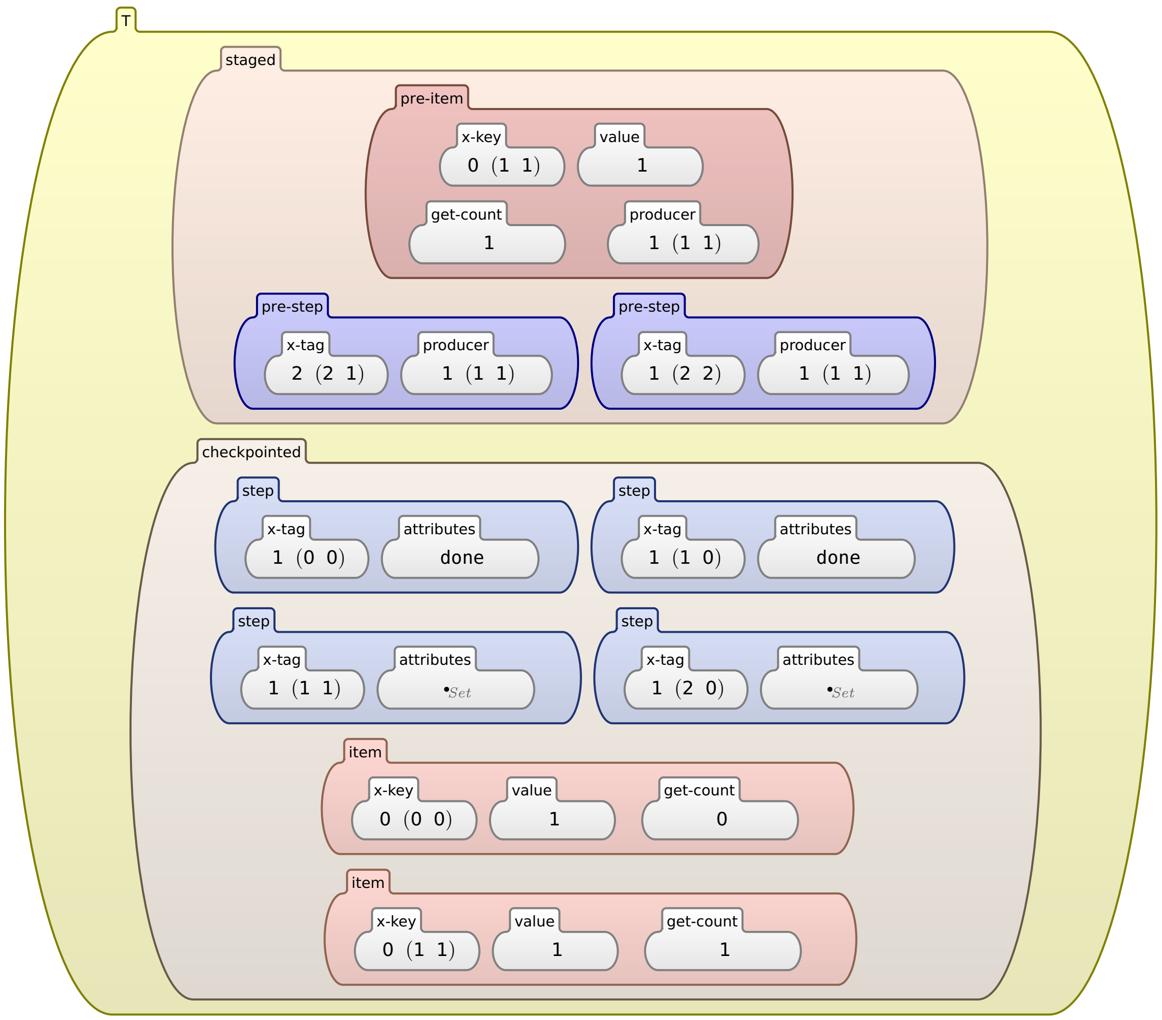
All state derivations are done completely asynchronously of the running computation graph, updating whenever new information arrives to the checkpoint process (without imposing any ordering constraints). The restart process for checkpoints generated by this modified model is identical to that described in algorithm 5.1 (with the assumption that all data not contained in the ⟨*checkpointed*⟩ cell is discarded). These two observations demonstrate that we have preserved the asynchronous nature of our checkpoint/restart system even with the addition of execution frontiers. It also follows that the modified CnC runtime maintains the ability to begin executing step instances while other step and item instances are still being restored from the checkpoint, as described in section 5.2.1.

7.3 Example: Restart with Pascal's Triangle

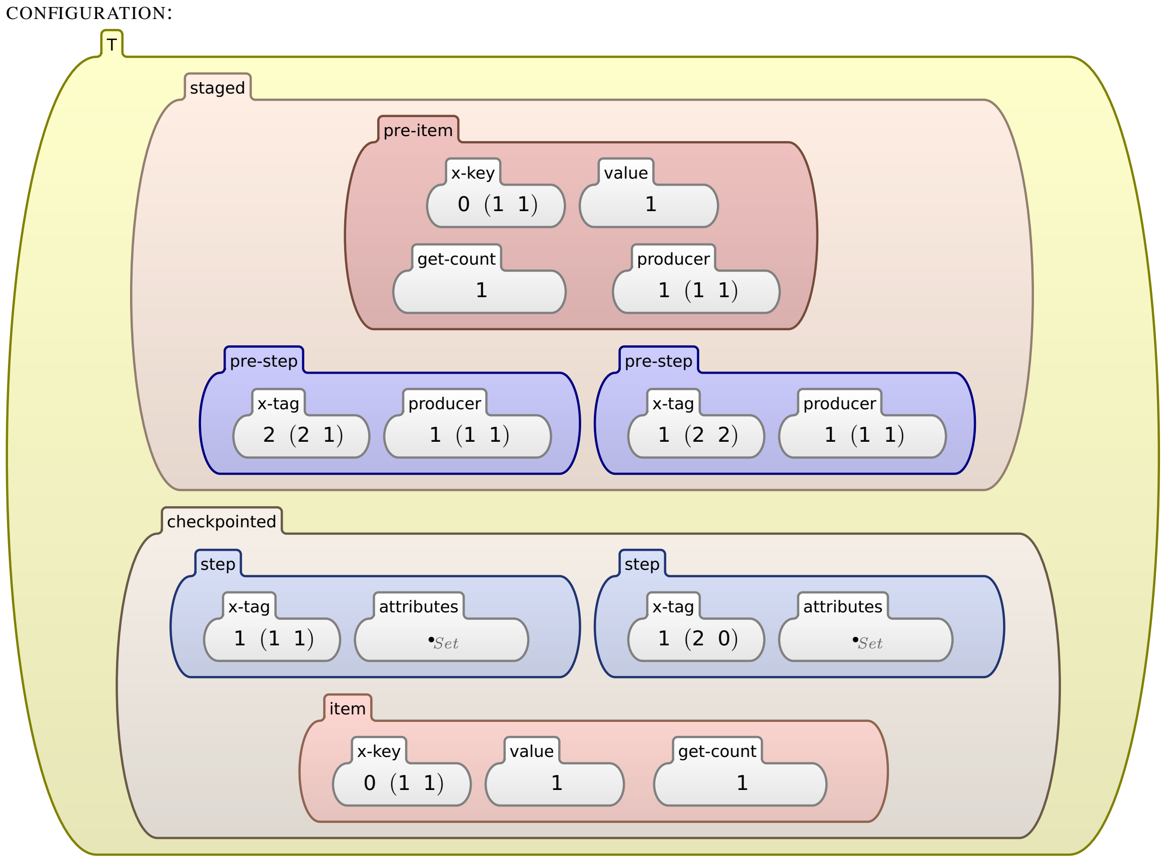
For this example we will again use the application for computing ${}_nC_k$. This application was described in section 3.2.1, and the source code is shown in listing 3.1. We consider an execution computing the value of ${}_2C_1$, and assume the same failure scenario described at the end of section 5.4. To review, ⦇edge-step: 0,0⦈, ⦇edge-step: 1,0⦈ and ⦇edge-step: 1,1⦈ have run and produced their outputs, but none of the step instances from row 2 have produced any outputs. ⦇edge-step: 0,0⦈ and ⦇edge-step: 1,1⦈ finished successfully, but ⦇edge-step: 1,0⦈ failed after producing its outputs.

Assume that all *put* and *prescribe* updates arrived from ⦇edge-step: 0,0⦈, ⦇edge-step: 1,0⦈ and ⦇edge-step: 1,1⦈. Also assume that the *step-done* update arrived for ⦇edge-step: 0,0⦈ and ⦇edge-step: 1,1⦈ instances, but not for ⦇edge-step: 1,0⦈ since it failed. In this case, as illustrated in figure 7.2, ⦇edge-step: 0,0⦈ and ⦇edge-step: 1,1⦈

CONFIGURATION:



(a) Checkpoint state before advancing the trailing edge.



(b) Checkpoint state after advancing the trailing edge.

Figure 7.2 : State of the CnC checkpoint for the Pascal's Triangle application execution scenario described in section 7.3. The state of the checkpoint is shown both before the trailing edge is applied to the checkpoint, as well as after the trailing edge removes from the checkpoint as many entries as possible. Recall from listing 3.1 that the integers 0, 1, and 2 are the unique IDs for the collections *pascal-entries*, *edge-step* and *inner-step*, respectively.

edge-step	inner-step	pascal-entries
$\langle 1, 0\rangle$		$\langle 1, 1\rangle \to 1$
$\langle 2, 0\rangle$		

Table 7.1 : Step and item instances added to the graph state by the environment during a sample restart of the ${}_2C_1$ calculation described in this section. Note that the step and item instances in this table correspond with the instances in the ‹*checkpointed*› cell in figure 7.2b, or the instances that have entered the checkpoint through the leading edge but have not yet been deleted by the advancement of the trailing edge.

are both *done* since all their outputs are accounted for. However, since the *step-done* message never arrived for (| edge-step: 1,1 |), the checkpoint cannot account for its outputs, and thus it is not considered *done* even though both of its outputs are actually present in the checkpoint.

Unlike the case described in section 5.4 using the simple checkpoint model, our updated model does not allow a step or item instance to enter the checkpoint until its producing step is *done*. This means that the ⟦ pascal-entries: 1,0 ⟧ and ⟦ edge-step: 2,0 ⟧ produced by ⟦ edge-step: 1,0 ⟧ remain in the ‹*staged*› cell, and are not considered part of the live execution frontier.

If we did a restart from this checkpoint, we would restore the step and item instances as outlined in table 7.1. In this case, the instance (| edge-step: 1,0 |) was restored, but neither of its outputs were restored. This restores the graph to a "clean" state with no duplicate data, thus satisfying the new restrictions for our model that were introduced in section 6.3. Since no outputs of partially-completed steps were included in the restored state, and all incomplete steps were restored, the execution will run to completion without creating any duplicate step or item instances.

7.4 Summary

We have described a model for computing checkpoints for a CnC graph execution based on asynchronous update messages from the CnC runtime while also tracking the state of the current execution frontier. By carefully tracking the liveness of step and item instances in the checkpoint state, we are able to ensure that no duplicate steps are run and no duplicate items are put after restoring a graph from a checkpoint. This conforms to the new restrictions on correctness of graph executions introduced in section 6.3.

Chapter 8

Checkpoint/Restart with CnC in Habanero-C

In this chapter, we introduce our implementation of checkpoint/restart within the existing Habanero-C CnC runtime (CnC-HC). We implement this checkpoint/restart system based on the model presented in chapter 7. We explain some of the changes necessary to the existing CnC-HC runtime to support C/R. We also give an example of using checkpoints to migrate a task between machines, and show some initial measurements of the overhead added by C/R.

8.1 Adding C/R Support to CnC-HC

As discussed in chapter 3, we can add support for C/R to an existing CnC runtime simply by adding hooks into the functions in the API layer between the user's CnC application and the CnC runtime. This is because all information necessary to create a checkpoint and restart from that checkpoint is encapsulated within the CnC graph state.

To allow multiple workers to send updates to the current checkpoint, we added a non-blocking queue (using liblfds [24]) for storing update messages. Data is encoded in Base64 (using libb64 [25]) before being written to disk. This adds some extra overhead, but having the data in ASCII form rather than binary form simplified the checkpoint processing. Currently a single thread dequeues entries and relays them to the checkpoint by writing them to a file on disk. In the future we would like to add support for multiple output threads, and communicate with a live checkpoint process rather than running the checkpoint processing algorithm only before a manual restart.

It is important to use the help-first policy [26] in HC when running CnC applications with C/R. This tells the runtime to prefer continuing the current task as opposed to executing new tasks immediately upon creation. If instead the work-first policy is used, then the runtime will eagerly execute newly-prescribed steps, possibly causing the environment to delay signaling that it has completed the graph initialization until the entire computation is complete. Parent steps that prescribe child steps would also be similarly delayed in signaling their completion.

8.1.1 C/R Hooks

Our checkpoint model needed input via the *put*, *prescribe*, and *step-done* commands, and our HC implementation needs to communicate information about those same things to the checkpointing process. Listing 8.1 outlines the hooks placed within the CnC-HC API functions. These represent the majority of calls to update checkpoint information. The `CNC_RUN` macro handles the logic of choosing whether to do a restart or run the user-provided graph initialization code. After running the initialization code, the macro adds a call to `cnc_cr_seed_done` to signal that the *cnc-env* step in the checkpoint is done by delivering its summary. The hooks in `CNC_PUT` and `CNC_PRESCRIBE` handle updates for item puts and step prescriptions. We had to add an additional hook at the end of the CnC-HC dispatcher's generated code for each step to send the step summary, corresponding to the *step-done* command. Finally, we included hooks to start and stop CnC checkpointing in `initGraph` and `deleteGraph`.

Appendix C includes a sample application that demonstrates the functions constituting the API layer containing our C/R hooks.

```
#ifdef CNC_CHECKPOINT

#define DO_MACRO_CONCAT(s1, s2) s1##s2
#define MACRO_CONCAT(s1, s2) DO_MACRO_CONCAT(s1, s2)
#define CNC_DUMMY_VAR MACRO_CONCAT(cnc_run_dummy_var_, __LINE__)

// NOTE: the for loop used here is just a hack to run cnc_cr_seed_done()
// after the user's code completes. Simple constant propagation eliminates
// the entire loop in gcc if compiled with -O1 or higher.
#define CNC_RUN \
  int CNC_DUMMY_VAR; \
  finish { CNC_DUMMY_VAR = cnc_cr_restart(); } \
  if (!CNC_DUMMY_VAR) \
      finish for (CNC_DUMMY_VAR=1; CNC_DUMMY_VAR--; cnc_cr_seed_done())

#define CNC_PUT(item, tag, coll, ctx) {\
  Put(item, tag, (ctx)->coll);\
  cnc_cr_put((CRStepCtx*)ctx, item, tag, #coll);\
}

#define CNC_PRESCRIBE(stepName, tag, ctx) {\
  prescribeStep(stepName, tag, ctx);\
  cnc_cr_prescribe((CRStepCtx*)ctx, stepName, tag);\
}

#else

#define CNC_RUN finish
#define CNC_PUT(item, tag, coll, ctx) Put(item, tag, (ctx)->coll)
#define CNC_PRESCRIBE(stepName, tag, ctx) prescribeStep(stepName, tag, ctx)

#endif /* CNC_CHECKPOINT */
```

Listing 8.1: Macros defining the CnC functions in HC. Two versions are declared: the first includes hooks for checkpoint/restart, whereas the second is the CnC-HC runtime without checkpoint/restart.

8.1.2 Checkpoint Message Handlers

Listing 8.2 shows some of the helper functions used to handle creating update messages that will be relayed to the checkpoint. These functions gather the necessary data and pack it all in a work item for the checkpoint-handler thread to process. Notice that most of these functions were used as our hooks in listing 8.1 from section 8.1.1. Listing 8.3 shows the main function for the checkpoint-handler thread. This thread simply dequeues work items, encodes the enclosed data in ASCII format, and then writes it out to the checkpoint file. This continues until the CnC runtime signals that execution is complete. If the graph execution completes before all checkpoint updates are handled, the thread still terminates since there is no longer a need to update the checkpoint in case of a restart.

8.1.3 Checkpoint Processing and Restarting

To process the messages written by the checkpoint-handler thread in the CnC-HC runtime, we re-implemented the process described by checkpoint the model in chapter 7 for asynchronously tracking the current execution frontier in the graph state. This implementation was done in Clojure, and is shown in listing 8.4. The result of running the checkpoint-processing code is an ASCII output representing the execution frontier that should be used for restart. This output can be passed as a file through the `CNC_RESTART` environment variable to the CnC-HC runtime, in which case the hooks placed within the `CNC_RUN` macro will cause a restart to take place rather than running the graph initialization code. We have tested this code with two CnC-HC applications: Cholesky factorization and Pascal's Triangle (calculating ${}_nC_k$). We were able to successfully restart a failed application. We were also able to simulate various failures by deleting portions of the checkpoint, and successfully

```c
void cnc_cr_put_gc(CRStepCtx *ctx, void *item, const char *key,
                   const char *coll, int gc) {
  if (gc > 0) { // No need to checkpoint unused data
    work_item *i = cnc_malloc(sizeof(work_item));
    work_item t = {
      .type = CNC_ITEM_T,
      .data = { .item={ .coll=coll, .key=key, .value=item, .get_count=gc } },
      .src_name=ctx->name, .src_tag=ctx->tag };
    *i = t;
    CNC_CR_PUSH_TAIL(work_queue, (void*)i);
    ++ctx->put_count;
  }
}

void cnc_cr_put(CRStepCtx *ctx, void *item,
                const char *key, const char *coll) {
  int gc = find_get_count(coll, key, (struct Context *) ctx);
  cnc_cr_put_gc(ctx, item, key, coll, gc);
}

void cnc_cr_prescribe(CRStepCtx *ctx, const char *step, const char *tag) {
  work_item *i = cnc_malloc(sizeof(work_item));
  work_item t = {
    .type = CNC_STEP_T,
    .data = { .step={ .tag=tag, .name=step } },
    .src_name=ctx->name, .src_tag=ctx->tag };
  *i = t;
  CNC_CR_PUSH_TAIL(work_queue, (void*)i);
  ++ctx->prescribe_count;
}

void cnc_cr_step_summary(CRStepCtx *ctx) {
  work_item *i = cnc_malloc(sizeof(work_item));
  work_item t = {
    .type = CNC_STEP_SUMMARY_T,
    .data = { .step_summary={
      .gets = ctx->gets, .put_count = ctx->put_count,
      .prescribe_count = ctx->prescribe_count
    } },
    .src_name=ctx->name, .src_tag=ctx->tag };
  *i = t;
  CNC_CR_PUSH_TAIL(work_queue, (void*)i);
}

void cnc_cr_seed_done() {
  cnc_cr_step_summary((CRStepCtx*)CNC_CR_GLOBAL_CONTEXT_PTR);
}
```

Listing 8.2: Helper functions for adding data to the checkpoint.

```
void *cnc_cr_main(void *unused) {
  work_item *i;
  const char **c;
  while (!STOP_WORK) {
    CNC_CR_POP_HEAD(work_queue, (void**)&i);
    if (i == END_SIGNAL) break;
    switch (i->type) {
      case CNC_ITEM_T:
        fprintf(cr_file, "%s{:type :item, :key [%s], :val \"",
          cr_startline(), i->data.item.key
        );
        output_base_64(i->data.item.coll, i->data.item.value);
        fprintf(cr_file,
          "\", :coll %s, :get-count %d,\n  :src {:name %s, :tag [%s]}}",
          i->data.item.coll, i->data.item.get_count,
          i->src_name, i->src_tag
        );
        break;
      case CNC_STEP_T:
        fprintf(cr_file,
          "%s{:type :step, :tag [%s], :name %s,\n"
          "  :src {:name %s :tag [%s]}}",
          cr_startline(), i->data.step.tag, i->data.step.name,
          i->src_name, i->src_tag
        );
        break;
      case CNC_STEP_SUMMARY_T:
        fprintf(cr_file,
          "%s{:type :step-summary, :tag [%s], :name %s,\n  :put-count %d,"
          " :prescribe-count %d,\n  :gets ( ",
          cr_startline(), i->src_tag, i->src_name,
          i->data.step_summary.put_count,
          i->data.step_summary.prescribe_count
        );
        if ((c = i->data.step_summary.gets)) {
          while (*c) {
            fprintf(cr_file, "{:coll %s, :key [%s]} ", c[0], c[1]);
            c += 2;
          }
        }
        fprintf(cr_file, ")}");
        break;
    }
    cnc_free(i);
  }
  fprintf(cr_file, (started ? ")\n" : "()\n"));
  fflush(cr_file);
  fclose(cr_file);
  return 0;
}
```

Listing 8.3: Main function for the checkpointing thread.

restart from those modified checkpoints. The full source code for these applications is available at http://habanero.rice.edu/vrvilo-ms.

8.2 Checkpoint Migration

The checkpoint format used in our implementation is simply a representation of the CnC graph state rather than a snapshot of some execution image state. This means a checkpoint can be migrated to a different machine and restarted on different hardware configuration and potentially a different operating system. The only requirement is that the target system must be able to restore the encoded data for each of the data item instances in the checkpoint. For example, we tested taking checkpoints of a CnC application on Rice's Davinci cluster (a 64-bit Linux system), a MacBook Pro running 64-bit OS X, and a 32-bit Linux desktop. On all three machines the application was restarted from all three checkpoints and produced the correct final result.

8.3 Initial Overhead Measurements

We measured the overhead that the C/R support code adds to a CnC application in the event that there is no failure. We did this measurement with the Cholesky factorization application, which is a tiled computation on a large $n \times n$ matrix. We ran these tests on a dedicated node of the Davinci cluster at Rice University. Each node has 12 Westmere processor cores at 2.83GHz, 4GB of RAM per core. We launched the CnC runtime with 8 worker threads, and when checkpointing support is enabled the runtime spawns an additional thread to handle checkpoint messages. These tests all ran the full set of checkpoint updates rather than terminating the process as soon as the calculation is complete. Figure 8.1 and table 8.1 summarize the test results.

```clojure
;; Namespace for checkpoint processing utility
(ns process-checkpoint
  (:require [clojure.java.io :as jio])
  (:import  [java.io PushbackReader]))

(defn read-file [file-path]
  (with-open [r (PushbackReader. (jio/reader file-path))]
    (binding [*read-eval* false] (read r))))

(defn sub-map [m & ks]
  (select-keys m ks))

;; Helper for "sanitize"
(defn sanitize-steps [staged checked]
  (for [[step-key step-val] staged
        :let [step (merge step-key step-val)]
        :when (and (= (-> step :puts count) (:put-count step))
                   (= (-> step :prescribes count) (:prescribe-count step))
                   (every? checked (:gets step))
                   (checked step-key))]
    (let [; remove step from "checked"
          checked (dissoc checked step-key)
          ; migrate puts and prescribes to "checked"
          checked (into checked
                    (concat
                      (for [p (:puts step)]
                        [(sub-map p :key :coll)
                         (sub-map p :val :get-count)])
                      (for [p (:prescribes step)]
                        [(sub-map p :name :tag) {}])))
          ; remove step from "staged"
          staged (dissoc staged step-key)
          ; update get counts
          up-get (fn [chk i]
                   (if (= 1 (get-in chk [i :get-count]))
                     (dissoc chk i)
                     (update-in chk [i :get-count] dec)))
          checked (reduce up-get checked (:gets step))]
      [staged checked])))

```

Listing continued on next page.

Listing continued from previous page.

```
;; This is the function that actually keeps the checkpoint state sane.
;; It goes through and migrates things from staged to checked, and
;; deletes things as they become dead.
(defn sanitize [staged checked]
  (loop [staged staged, checked checked]
    (if-let [res (seq (sanitize-steps staged checked))]
      (let [[[staged checked]] res]
        (recur staged checked))
      [staged checked])))

(def initial-checked {{:name :cnc-env :tag []} {}})

;; Main loop for processing messages from the runtime. You could think
;; of this as running real-time, being fed a stream of these messages.
(defn process-msgs [msgs]
  (loop [[m & msgs] msgs, [staged checked] [{} initial-checked]]
    (if (nil? m) {:staged staged :checked checked}
      (case (:type m)
        :step
        (let [step (sub-map m :name :tag)
              staged (update-in staged [(:src m) :prescribes] conj step)]
          (recur msgs (sanitize staged checked)))
        :item
        (let [item (sub-map m :key :val :coll :get-count)
              staged (update-in staged [(:src m) :puts] conj item)]
          (recur msgs (sanitize staged checked)))
        :step-summary
        (let [step (sub-map m :name :tag)
              summary (sub-map m :put-count :prescribe-count :gets)
              staged (update-in staged [step] merge summary)]
          (recur msgs (sanitize staged checked)))))))

(defn output-for-restart [{checked :checked}]
  (when (not= checked initial-checked)
    (doseq [x checked :let [x (apply merge x)]]
      (condp deliver x
        :key (do (apply println "I" (:coll x)
                        (count (:val x)) (:get-count x) (:key x))
                 (println (:val x)))
        :tag (apply println "S" (:name x) (:tag x))))))

;; Read in the checkpoint file (path in command-line-args) and process it
(let [[chkpt-file-path] *command-line-args*
      res (process-msgs (read-file chkpt-file-path))]
      (output-for-restart res))
```

Listing 8.4: Clojure code for processing CnC-HC checkpoint output.

For input matrices of up to 4 million entries the completion time for the base case (without C/R support) and the case with Base64-encoded C/R data are almost identical. However, as we continue to increase the amount of data involved in the calculation, the C/R configuration with encoded data takes increasingly longer to complete. The gap between these two configurations grows, and as seen in the Ratio column of table 8.1, the gap seems to be increasing super-linearly. We theorized that the Base64 encoding of the CnC item data during the checkpointing process, and thus we included test results for checkpointing the raw binary data of the CnC item values. The only difference with this implementation is that rather than encoding the item values in Base64 before writing them to the checkpoint file, we write the raw bytes directly. When the Base64 conversion was removed, we see that the run times with and without checkpointing are almost identical.

As seen in the last column of table 8.1, writing the raw binary values to the checkpoint file only added an overhead of about 1% on average to the total running time. Although our implementation still uses the Base64 encoding for simplicity in processing the checkpoint files, we believe these results show that this technique can be optimized such that the checkpointing overhead is very minimal. However, an imbalance between the amount of data produced by a step and the amount of computation done within that step can lead to a noticeable overhead due to checkpointing. This is illustrated in figure 8.2 in the cases with 25×25 and 50×50 tiles, which are too small to produce a sufficient amount of computation within the steps. These results imply that the additional I/O overhead incurred for checkpointing should be minimal so long as the application does not produce data in excess of the system's maximum I/O rates. In the case when the application is producing data faster than the hardware supports writing it, the overhead will increase along with the I/O backlog.

Entries	Base	C/R Encoded	Ratio	C/R Raw	Ratio
1M	0.06s	0.06s	1.01	0.07s	1.03
4M	0.38s	0.38s	1.00	0.39s	1.01
9M	1.24s	1.74s	1.40	1.25s	1.01
16M	2.90s	9.07s	3.13	2.93s	1.01
25M	5.62s	21.19s	3.77	5.68s	1.01

Table 8.1 : Time data corresponding to the means plotted in figure 8.1. The *Ratio* columns are the ratios of the checkpoint/restart running times to the base running times. Each time measurement is the average of five separate runs for each given configuration.

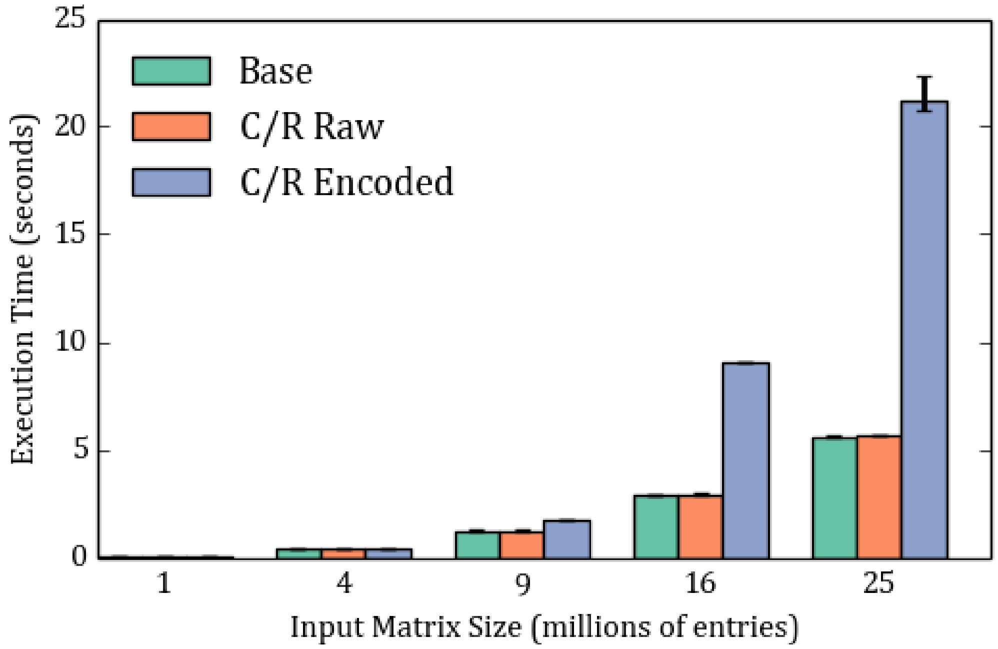


Figure 8.1 : Total running time for CnC-HC Cholesky, with increasing input matrix sizes. The three bars at each point correspond to (from left to right) running without checkpointing enabled (Base), with checkpointing enabled plus Base64 encoding (C/R Encoded), and with checkpointing of raw binary data (C/R Raw). Each bar shows the average time for five runs, with the error bars denoting the maximum and minimum observed run times. Table 8.1 shows an alternate analysis of this data.

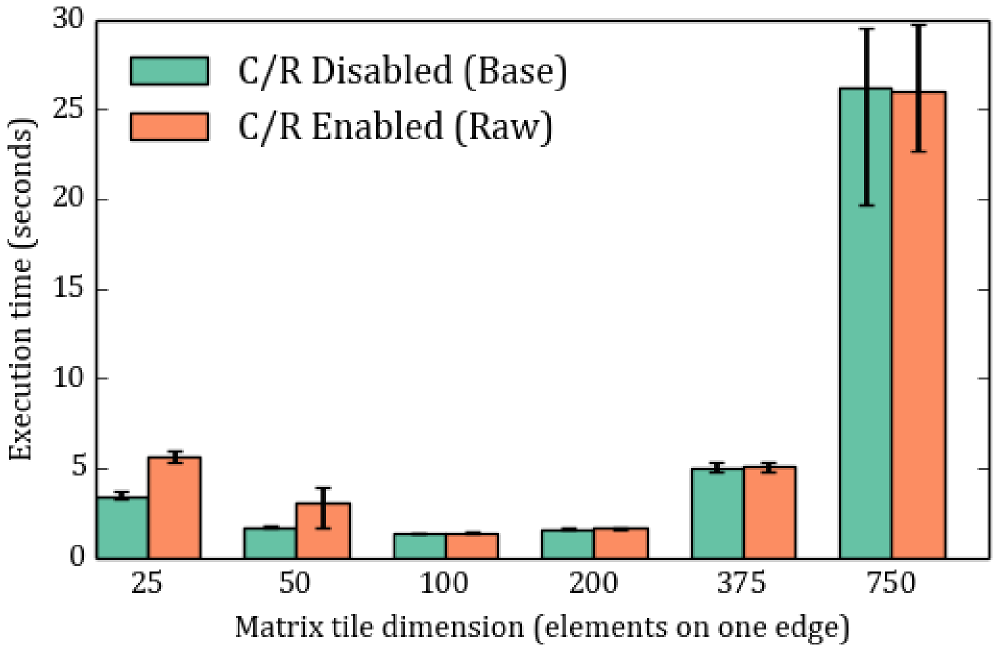


Figure 8.2 : Total running time for CnC-HC Cholesky, varying the tile size used on a 3000×3000 matrix (9 million elements). Each bar shows the average of 10 runs with the given configuration, with the error bars denoting the minimum and maximum observed run times. The C/R-enabled runs are using raw binary for the checkpoint data encoding rather than Base64. The first two tile sizes shown are too small to produce a sufficient amount of work per step to hide the checkpoint I/O overheads, and therefore the C/R-enabled times are significantly higher than the C/R-disabled times at these points. In the case of a 25×25 tile, the work per step is not even sufficient to hide the runtime overhead for managing steps, and therefore the C/R disabled time is also higher at this point than at the next three points, which are more optimal tile sizes for this input matrix size. The last two tile sizes are too large, resulting in insufficient parallelism and leading to longer running times.

8.4 Summary

We added C/R support to the existing CnC-HC implementation. The additions were fully encapsulated within the CnC-HC runtime so as to not effect the user's application code. We discussed the ability to migrate checkpoints between machines with different hardware and OS configurations. Finally, we analyzed some initial performance results to measure the overhead incurred by supporting C/R when no error occurs.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

Checkpointing requires no synchronization between execution graph and checkpoint process. The CnC checkpoint always contains a valid state regardless of the order in which updates arrive. All of our C/R hooks are within the CnC runtime, allowing us to make the C/R support completely transparent to the CnC application writer. In other words, an application developer can simply enable C/R and run the same program (reusing the same step code and environment). We demonstrated these properties with two implementations of C/R in CnC. The first was built with an executable model written in K, while the applications used a thin API written in Clojure. The second was an implementation of C/R within the existing CnC-HC runtime. With the CnC-HC runtime, we also demonstrated the ability to migrate checkpoints between systems with different hardware configurations and operating systems. We also identified some possible targets for optimization to decrease the overhead added by the C/R support code.

9.2 Future Work

Although the model from chapter 7 maintains a sane execution frontier at all times, it potentially throws away useful information from the staging area in the event of a restart. By tracking unique step tags of get operations on each item—and only decrementing the get count for unique gets—we could maintain the idempotent and monotonic properties of the runtime. This would also allow us to relax some of the restrictions in the chapter 7 checkpoint model, and potentially allow us to include within the live portion of the checkpoint a larger set of the data received by the checkpoint.

As discussed in section 8.3, the high rates of data production in a CnC application could limit the usefulness of our C/R technique, as it could result in an I/O backlog. However, in some applications it may be possible to partition a graph in such a way that only some fraction of the step and item instances need to be included in the checkpoint. This would enable us to exclude intermediate instances from the checkpointing process, and thus reduce the total required I/O footprint. This would require the ability to create partitions within the CnC graph, such that a group of step instances is considered as a single step instance to the checkpoint. We would like to further explore the possibilities and implications of such graph partitions.

In section 2.1.1 we noted that memory management mechanisms must be carefully adapted to maintain correctness in the presence of checkpointing. The CnC-HC runtime used in this thesis does not include any mechanisms memory management. We would like to implement the C/R technique described in this paper within a CnC runtime that handles memory management of data item instances.

Finally, this work was limited in application to a shared-memory environment. However, we believe that many of the concepts in this thesis could be applied to a more general distributed CnC runtime. We would like to explore what additional requirements would need to be enforced to maintain distributed checkpoints that could be recombined and restarted in the event of a failure.

Bibliography

[1] J. Daly, B. Harrod, T. Hoang, *et al.*, "Inter-Agency Workshop on HPC Resilience at Extreme Scale," *National Security Agency Advanced Computing Systems, Februrary*, 2012.

[2] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11, IEEE, 2010.

[3] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *IEEE International Conference on Cluster Computing*, pp. 93–103, IEEE, 2004.

[4] M. Hall, R. Lethin, K. Pingali, *et al.*, "ASCR Programming Challenges for Exascale Computing," tech. rep., U.S. DOE Office of Science (SC), July 2011.

[5] M. Burke, K. Knobe, R. Newton, and V. Sarkar, "Concurrent collections programming model," in *Encyclopedia of Parallel Computing* (D. Padua, ed.), pp. 364–371, Springer US, 2011. http://goo.gl/UF4L0I.

[6] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3, pp. 203–217, 2010.

[7] D. Sbîrlea, K. Knobe, and V. Sarkar, “Folding of tagged single assignment values for memory-efficient parallelism,” in *Euro-Par 2012 Parallel Processing*, pp. 601–613, Springer, 2012.

[8] Z. Budimlic, A. M. Chandramowlishwaran, K. Knobe, G. N. Lowney, V. Sarkar, and L. Treggiari, “Declarative aspects of memory management in the concurrent collections parallel programming model,” in *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pp. 47–58, ACM, 2009.

[9] E. W. Weisstein, “Binomial Coefficient. From *MathWorld*–A Wolfram Web Resource.” http://mathworld.wolfram.com/PascalsTriangle.html. Accessed: 2014-03-15.

[10] E. W. Weisstein, “Pascal’s Triangle. From *MathWorld*–A Wolfram Web Resource.” http://mathworld.wolfram.com/BinomialCoefficient.html. Accessed: 2014-03-15.

[11] P. H. Hargrove and J. C. Duell, “Berkeley lab checkpoint/restart (BLCR) for Linux clusters,” in *Journal of Physics: Conference Series*, vol. 46, p. 494, IOP Publishing, 2006.

[12] K. M. Chandy and L. Lamport, “Distributed snapshots: determining global states of distributed systems,” *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[13] J. Ansel, K. Arya, and G. Cooperman, “DMTCP: Transparent checkpointing for cluster computations and the desktop,” in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, IEEE, 2009.

[14] W. Ma and S. Krishnamoorthy, "Data-driven fault tolerance for work stealing computations," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 79–90, ACM, 2012.

[15] K. Knobe and C. D. Offner, "TStreams: How to write a parallel program," Tech. Rep. HPL-2004-193, HP Labs, 2004. http://www.hpl.hp.com/techreports/2004/HPL-2004-193.pdf.

[16] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 712–725, IEEE, 2013.

[17] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-C." http://habanero.rice.edu/hc.

[18] S. Tasirlar and V. Sarkar, "Data-driven tasks and their implementation," in *International Conference on Parallel Processing*, pp. 652–661, IEEE, 2011.

[19] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010. http://www.kframework.org/.

[20] R. Hickey, "The Clojure programming language," in *Proceedings of the 2008 symposium on Dynamic languages*, p. 1, ACM, 2008.

[21] "Habanero CnC." http://habanero.rice.edu/cnc.

[22] Traian Florin Şerbănuţă, "Reading tuples in K." K-user mailing list, November 2012. http://lists.cs.uiuc.edu/pipermail/k-user/2012-November/000262.html.

[23] K. Knobe, “Ease of Use with Concurrent Collections (CnC),” in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar’09, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2009.

[24] T. Douglass, “liblfds: A portable lock-free data structure library, written in C.” http://www.liblfds.org/.

[25] C. Venter, “libb64: ANSI C Base64 Encoding/Decoding Routines.” http://libb64.sourceforge.net/.

[26] Y. Guo, R. Barik, R. Raman, and V. Sarkar, “Work-first and help-first scheduling policies for async-finish task parallelism,” in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, IEEE, 2009.

Appendix A

CnC Sans Control Collections

Control collections are traditionally used in CnC to help abstract away the creation of new step instances. In the traditional model, a CnC step can only create a new step instance indirectly by *putting* a control tag into a control collection, which in turn causes a new step instance to be *prescribed* in each step collection driven by that control collection [6]. Figure A.1 illustrates the graph structure of a simple application implementing a two-level filter (originally shown in figure 2.1) in the CnC model with control collections. Although this is a useful abstraction for reasoning about some programs, in practice we have found that the concept of control collections tends to confuse new users. Furthermore, we have found that in the majority of our CnC applications there is always a single step collection associated with each control collection. These observations are reflected by the absence of control collections in the Habanero variants of CnC, developed at Rice University [21]. For these reasons, we choose to model a simplified CnC that uses only item and step collections.

We now demonstrate that this simplified model can still be applied to general CnC applications. In the case that the control-collection/step-collection relationship is a bijection, we can safely substitute each control-tag *put* with the equivalent step *prescribe* while maintaining identical program behavior. In the rare case that a control collection drives multiple step collections, each *put* must be replaced by one *prescribe* per associated step collection. These two transformations enable us to transform any general CnC application to one without control collections; therefore, we know

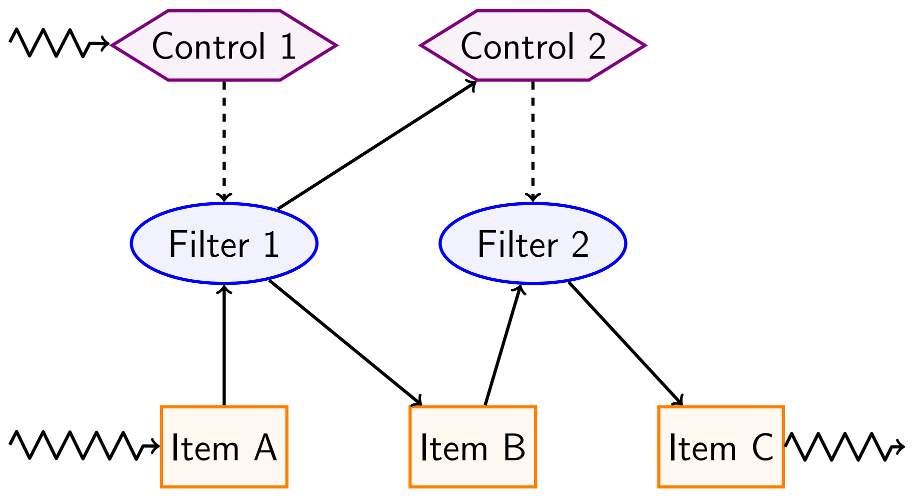


Figure A.1 : The CnC graph representation of a simple data filtering application (originally shown in figure 2.1), with the addition of control collections. The two control collections, which drive the two step collections, are represented by the hexagonal nodes. Solid edges ending at a control collection represent a put of a tag to that control collection. In this version of the CnC model, only control collections can prescribe step instances.

that modeling only the step and item collections of CnC is sufficient to describe CnC applications in general.

Our model of CnC has no explicit control collections. Instead, we model only the step and item collections, as described above. This choice reflects the design of existing CnC implementations designed at Rice University [21].

Appendix B

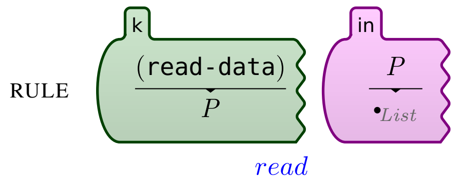
Details of the Executable Model's I/O

The full details of I/O between the Clojure wrapper and the K model are complicated by the fact that K cannot fully parse data coming from *stdin*[1]; therefore, we chose to encode our input as a stream of integers, and decode the integers into more a more readable representation via rewrite rules.

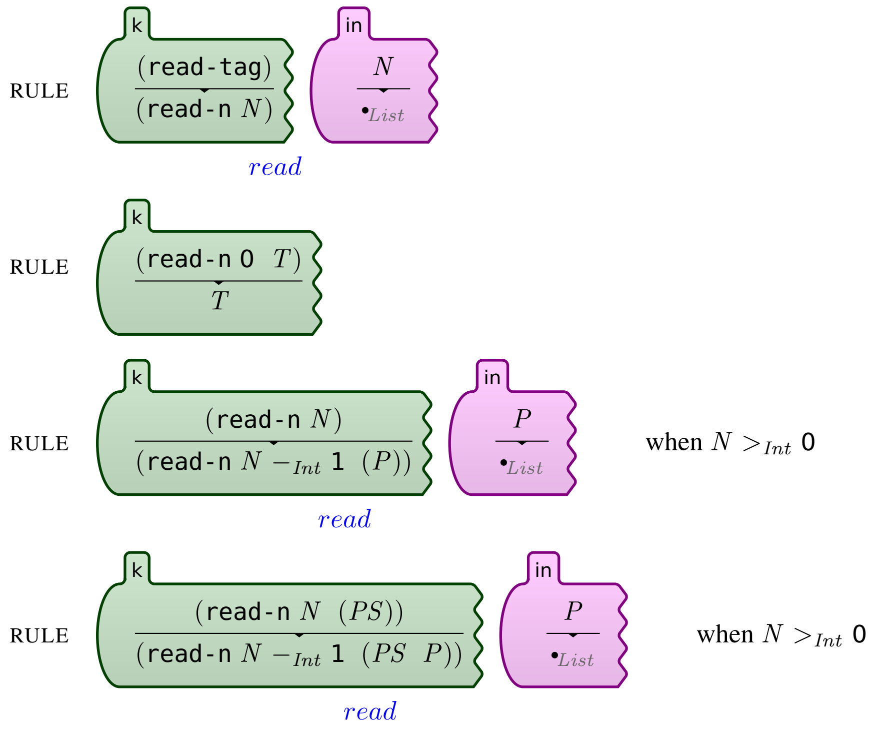
Rule set B.1 contains the rewrite rules for reading data (integers) and tags (tuples) from *stdin* as an example of this decoding process. The `read-data` command is relatively straightforward and can be implemented using a single rewrite rule that copies an integer from *stdin*. However, the `read-tag` command requires a set of 4 rewrite rules working together to incrementally read the tag components one by one from *stdin*.

The full source for all four K models presented in this thesis—along with the corresponding graphical representations—are available at http://habanero.rice.edu/vrvilo-ms.

[1] See *Reading tuples in K* on the K-user mailing list: http://lists.cs.uiuc.edu/pipermail/k-user/2012-November/000262.html.



(a) Reading data.



(b) Reading tags.

Rule Set B.1: Rewrite rules for decoding the integer-stream representations of CnC data items (integers) and tags (integer tuples) read from *stdin*.

Appendix C

A Sample CnC-HC Application

Listings C.1 to C.3 contain the graph initialization and step code for a sample CnC-HC application. This application computes binomial coefficients using Pascal's Triangle, as described in section 2.1.2. The CnC-HC source code included here corresponds closely with the Clojure code from listing 3.1.

The full source code for both the Pascal's Triangle and the Cholesky factorization examples are available online at http://habanero.rice.edu/vrvilo-ms.

```c
#include "Common.h"

static int ONE_CELL = 1;

void edgeEntryStep(int row, int col, Context *cnc_ctx) {
  // Put the value 1 for the entry at <row, col>
  char *tagEntry1 = CREATE_TAG(row, col);
  CNC_PUT(&ONE_CELL, tagEntry1, entry, cnc_ctx);

  // All but the last row will prescribe a step in the next row
  if (row < cnc_ctx->n) {
    char *tagEdgeEntry2 = CREATE_TAG(row+1, col);
    char *stepName = col ? "innerEntryStep" : "edgeEntryStep";
    CNC_PRESCRIBE(stepName, tagEdgeEntry2, cnc_ctx);
    // Right-edge entries also prescribe the last entry in the next row
    if (col == row) { // Left edge
      char *tagEdgeEntry3 = CREATE_TAG(row+1, col+1);
      CNC_PRESCRIBE("edgeEntryStep", tagEdgeEntry3, cnc_ctx);
    }
  }
}
```

Listing C.1: CnC-HC step code for computing the entries along the left and right edges of Pascal's Triangle, which always contain the value 1.

```c
#include "Common.h"

void innerEntryStep(int row, int col, int entry0, int entry1, Context *cnc_ctx) {
  // Sum the two elements above the entry <row, col>
  int *entry2;
  entry2 = malloc(sizeof(int));
  *entry2 = entry0 + entry1;

  // Put the sum of the parent elements as the entry at <row, col>
  char *tagEntry2 = CREATE_TAG(row, col);
  CNC_PUT(entry2, tagEntry2, entry, cnc_ctx);

  // All but the last row will prescribe a step in the next row
  if (row < cnc_ctx->n) {
    char *tagInnerEntry3 = CREATE_TAG(row+1, col);
    CNC_PRESCRIBE("innerEntryStep", tagInnerEntry3, cnc_ctx);
  }
}
```

Listing C.2: CnC-HC step code for computing the inner entries of Pascal's Triangle.

```c
#include ”Dispatch.h”
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
  int *n = (int*) malloc(sizeof(int)*3);
  int *k = n+1;
  int **result = malloc(sizeof(int*));
  *result = n+2;
  // Read arguments
  sscanf(argv[1], ”%d”, n);
  sscanf(argv[2], ”%d”, k);
  // Init graph
  Context *cncGraph = initGraph();
  cncGraph->n = *n;
  cncGraph->k = *k;
  // Run
  CNC_RUN {
     tag = CREATE_TAG(0, 0);
     CNC_PRESCRIBE(”edgeEntryStep”, tag, cncGraph);
  }
  // Get result
  tag = CREATE_TAG(*n, *k);
  CNC_GET((void**)result, tag, cncGraph->triangle, NULL);
  printf(”%d choose %d = %d\n”, *n, *k, **result);
  // Cleanup
  deleteGraph(cncGraph);
  return 0;
}
```

Listing C.3: Graph initialization code for the Pascal’s Triangle application (computing ${}_nC_k$ via the triangle entries) in CnC-HC.