The Eight Fundamental Patterns of Concurrent Collections

World Wide Web: http://www.intel.com

Disclaimer and Legal Information

Revision History

Revision Number	Description	Revision Date
1.0	Initial release.	June, 2008
2.0	Update for 0.4 release	September, 2009
3.0	Update for 0.5 release	March, 2010
4.0	Less dependent on release; update for 0.7 release	January, 2012

Contents

List of Figures

1 Introduction

This document presents the eight basic design patterns of Intel® Concurrent Collections for C++. These eight design patterns are the eight ways that any two steps of computation might relate to each other. For each of the patterns we show at least one example including all the code required and a brief discussion.

One of goals of this language is to separate the concerns of the domain-expert who is focused on the semantics of the application from the tuning-expert who is focused on performance. The topics addressed in this document address the needs of the domain-expert.

Below the terms language, graph specification and program refer to the Intel® Concurrent Collections for C++ language, the Intel® Concurrent Collections for C++ graph specification and an Intel® Concurrent Collections for C++ program.

1.1 The problem

Many languages for expressing parallelism embed parallelism within serial code. Serial code requires a serial ordering. If there is no semantically required ordering, an arbitrary ordering must be specified. Execution of the code in parallel often requires a complex analysis to determine if reordering is possible. Overwriting of memory locations makes this analysis difficult. These two characteristics of serial code, overwriting and arbitrary serialization, combine to make it difficult to uncover alternate valid executions either manually or automatically. For these reasons, embedding parallelism in serial code can limit both the language's effectiveness and its ease of use.

In this language, programs are written in terms of high-level application-specific operations. They are partially ordered according to their semantic constraints only. In addition, the data that flows among these operations is by value, not by location. At the level of these high-level operations, there is no overwriting and no arbitrary serialization.

This approach supports an important separation of concerns. There are two roles involved in writing a parallel program. One is the domain expert, the developer whose interest and expertise is in finance, genomics, etc. The other is the tuning expert, whose interest and expertise is in performance. These may be distinct individuals or the same individual at different stages in application development. The tuning expert may in fact be software (static compiler analysis or dynamic runtime analysis). This language separates expression of the semantics of the computation (the work of the domain expert) from the expression of the actual parallelism, scheduling and distribution for a specific architecture (the work of the tuning expert). This separation simplifies the work of the domain expert. Writing in this language does not require any reasoning about parallelism or any understanding of the target architecture. The domain expert is concerned only with her area of expertise (the semantics of the application). The tuning expert is given the maximum possible freedom to map his computation onto the target architecture. In this document we will discuss all the topics relevant to the domain expert.

1.2 Lanugage concepts via an example

We will use face detection as an example application to describe the language. Detection is performed on a sequence of images. Each image is further subdivided into square sub-images (called *windows*) of any size and at any position within the image. Each window is processed by a sequence of classifiers. If any classifier in the sequence fails, the window does not contain a face and the remainder of the classifiers need not process that window. The goal of this approach is to rapidly reject any window not containing a face.

A program is specified by a graph with three types of nodes (steps, items and tags) and three types of edges (producer relations, consumer relations and prescription relations). We will introduce the language by showing the process one might go through to create a version of the face detector in this language. This discussion refers to Figure 1-1: Face Detection: CnC specification which shows a simplified graphical representation of our face detection application.

1.2.1 Creating a graph specification

1.2.2 Determine the steps

The computation is partitioned into high-level operations called step collections. Step collections are represented as ovals. In this application, the step collections are the classifiers C1, …, Cn. We use the term step collection to indicate that it is a collection composed of distinct step instances where steps are the unit of execution.

1.2.2.1 Determine items, producer relations, consumer relations

Similarly, the data structures are partitioned into data structures called *item collections.* Item collections are represented by rectangles. In this application there is only one item collection, image. We use the term item collection to indicate that it is a collection composed of distinct item instances where Item instances are the units of storage, communication and synchronization. The producer and consumer relationships between step collections and item collections are explicit. The consumer relationships are represented as directed edges into steps. Producer relations are represented as directed edges out from steps. The image items are consumed by the classifier steps. There are no items produced in this application.[1]

The environment (the code that invokes the graph) may produce and consume items and tags. These relationships are represented by directed squiggly edges. In our application, for example, the environment produces Image items.

After these first two phases we have a description that is typical of how people communicate informally with one another at a whiteboard. The next two phases are required to make this informal description precise enough to execute.

1.2.2.2 Determine tag components

The computations represented by circles are not long-lived computations that continually consume input and produce output. Rather, each step collection, represents a set of individual step instances and each item collection, represents a set of individual item instances. The execution of steps and the communication or synchronization of items is performed on instances.

We need to distinguish among the instances of a step collection and instances of an item collection. Each dynamic instance of a step or an item is uniquely identified by an application-specific tag. A *tag component* might indicate a node identifier in a graph, a row number in an array, an employee number, a year, etc. A complete *tag* might be composed of several components, for example, employee number and year or maybe xAxis, yAxis, and zAzis.

In our example, the instances of the image collection are distinguished by image#. The classifier step instances are distinguished by image# and window# pair.[2] Notice that, in this example, a classifier step inputs the whole image even though it operates only on one window within the image.

1.2.2.3 Determine the tag collections and prescriptive relations

Knowing the tag components that allow us to distinguish among instances is not quite enough to bring us to a specification that is precise enough to execute. Knowing that we distinguish instances of classifier1 by values of image# and window# doesn't tell us if classifier1 is to be executed for image#2873, window#56. We have already introduced item collections for data and step collections for computation. Now we introduce *tag collections* for control.

[1] The directed edges from steps to the triangles are discussed below.
[2] The image tag component is simply an int that identifies a specific image. The window tag component in a real version is actually more complex than a simple int. For example, one way to identify a window uses three components, an x and y coordinate (indicating the top left corner of the window) and a length. Tag components can be of user-defined data types. There are some restrictions in the initial release. Please see the latest documentation for details.

Tag collections, sets of tag instances, provide the control mechanism. Tag collections are shown in triangles. The tag collections in this graph are T1, … , Tn . A *prescriptive relation* may exist between a tag collection T and a step collection S. The meaning of such a relationship is this: if a tag instance t, say image# 2873, window# 56, is in T, then the step instance s in S with tag value t, image# 2873, window# 56, will execute. A prescriptive relation is shown as a dotted edge between a tag collection and a step collection. A step collection S prescribed by a tag collection T must have tags of the same form as tags in T. Thus we know the form of the tags for the classifiers.

Usually control flow indicates not only *if* code executes but also *when*. In the language, the control via tags only indicates *if* code executes. *When* it executes is up to a subsequent scheduler.

When we add a tag collection to our specification, we have to add the corresponding producer relation. For example, the environment produces T1 which indicates all the windows for all the images. The point of step collection C1 is to determine that many of these windows definitely do not contain a face and which might contain a face. An instance of C1, say with tag image# i and window# w, will either produce tag T2 with

tag image# i and window# w (indicating that it might be a face) or it will produce nothing (indicating that it is definitely not a face). So step collection C1 produces tag collection T2. The tag instances in T2 determine which instances of C2 will execute. Similarly other step collections and tag collections have producer relationships.

At this point the importance of tags should be clear. Tags make this language more flexible and more general than a streaming language. In addition, the tag mechanism separates the question of *if* a step will execute from *when* a step will execute. The domain-expert determines *if* a step will execute. The tuning-expert determines *when* it will execute. This separation allows for more effective tuning.

The resulting graph is shown in Figure 1-1: Face Detection: CnC specification.

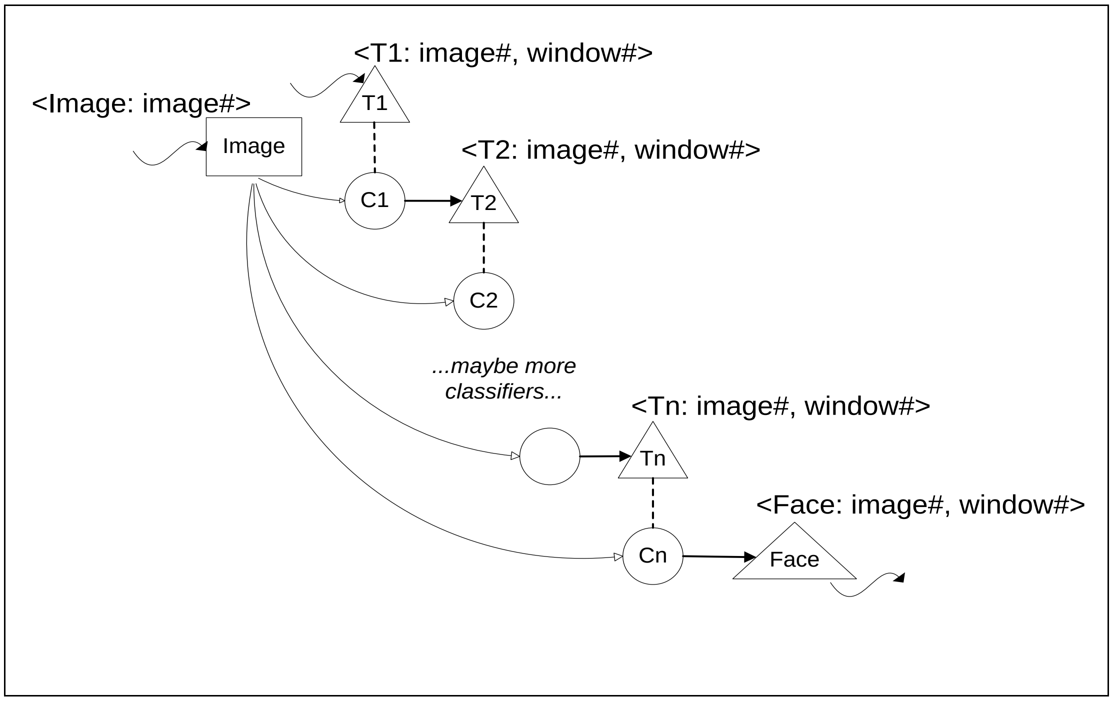


Figure 1-1: Face Detection: CnC specification

This graph can also be represented in a textual syntax:

```
// I/O
env -> [image: image#];
env -> <T1: image#,window#>;
<face: image#,window#> -> env;

// Prescription relations
<T1: image#,window#> :: (C1: image#,window#);
<T2: image#,window#> :: (C2: image#,window#);
 …
<Tn: image#,window#> :: (Cn: image#,window#);

// Consumer relations
[image: image#] -> (C1: image#,window#);
[image: image#] -> (C2: image#,window#); …
[image: image#] -> (Cn: image#,window#);

// Producer relations
(C1: image#,window#) -> <T2: image#,window#>;
(C2: image#,window#) -> <T3: image#,window#>;
 …
(Cn: image#,window#) -> <Face: image#,window#>;
```

1.2.3 Coding the steps (high-level operations)

In addition to specifying the graph, we need to code the steps in a serial language. The step has access to the values of its tag components. It uses **`get`** operations to consume items and **`put`** operations to produce items and tags. Please refer to the API documentation for more details.

1.2.4 Parallelism

The vision expert going through the process above needs to know a lot about vision but nothing in the process involves any reasoning about parallelism. The resulting program makes the constraints on parallelism explicit, but not the parallelism itself. The constraints are either data dependences (steps produce items that are consumed by other steps) or control dependences (steps produce tags that prescribe other steps). This simple example only contains control dependences.

The graph shows relationships among collections, i.e., it indicates that between item collection [I] and step collection (S) there either is or is not a consumer (or producer) relation. To understand the constraints on parallelism we need more specifics about the relations among instances. *Tag functions* provide this information. In our example, the producer tag function that maps the tag of a classifier step, say (C1) to the tag of the tag collection <T2> is the identity function, e.g.., (C1: i, w) can only produce <T2: i, w> not <T2: i+1, w> for example.

Other applications, for example nearest neighbor computations, have more interesting tag functions. The domain expert may provide tag functions (for better performance) or not (for better ease-of-use). We will not discuss tag functions further in this document. What is important is that tag functions require only domain knowledge, not understanding of parallelism. Given this tag function, the tuning expert or the runtime can determine the constraints. This determination does not involve heavy analysis because there is no overwriting and no arbitrary serialization.

In this example, there are no constraints among images and there are no constraints among windows in an image. The only constraint is that for a given window w of a given image i, the classifiers are executed in order because, for example, we don't know if classifier (C2: i, w) will execute until (C1: i, w) completes.

1.2.5 Rules

1.2.5.1 Rules for Individual collections and for the environment

TAG COLLECTIONS

- A tag collection may prescribe multiple step collections.
- Each step collection is prescribed by exactly one tag collection.

- Although a tag collection might be produced by a step collection, it doesn't make sense for it to be consumed by one.

STEP COLLECTIONS

- A step collection may consume multiple item collections.
- A step may not consume any items. It may operate only on its tag.
- A step collection may produce multiple item collections and tag collections.

ITEM COLLECTIONS

- An item collection may be consumed by multiple step collections.
- An item collection may be produced by multiple step collections.

THE ENVIRONMENT

- The environment may produce multiple tag collections and item collections.
- The environment may consume multiple tag collections and item collections.

1.2.5.2 Rules for whole graph specifications

QUESTIONABLE PRACTICES

The following situations indicate that the program is either incomplete or has irrelevant parts. We may provide warnings for some of these situations.

- A program should input at least one tag collection from the environment.
- A program should output at least one collection (either tag or item) to the environment.
- Each item collection produced should be consumed (possibly by the environment).
- Each tag collection produced (possibly by the environment) should be used to prescribe some step collection or it should be consumed by the environment.
- Each item collection produced (possibly by the environment) should be consumed (possibly by the environment). However, an item collection should not be produced only by the environment and consumed only by the environment.

REASONABLE PRACTICES

- A graph need not be connected.
- It is not necessary for a step collection to input item collections. A step may operate on its tag value only.
- It is not necessary for a program to input any item collection from the environment. It may operation only in input tag collections.

1.2.5.3 Rules for valid step code

IN GENERAL

- A step may not reference (read or write) any global values.

  - it accesses global state only by reading its tag or by getting items

  - it modifies the global state only by putting items or tags

- Gets and puts in the steps must be consistent with the associated program graph
- The steps adheres to dynamic single assignment (during execution, the program never generates multiple instances of an item with the same name and tag but different contents.)

IN THE CURRENT IMPLEMENTATION

- All gets must occur before any puts
- All gets must occur before any heap memory allocations
- A step can not use the iterator to get all instances in a collection. The iterator is for use by the environment only.

1.2.5.4 Rules for valid environment code

- The environment code can be a significant part of the application or it can simply act as a driver for the graph program.
- It may invoke multiple instances of a given graph.
- It may invoke distinct graphs.
- For each invocation:

  - The environment performs puts of tags and items into the graph.

  - Then it waits for all steps to finish. The execution has access to the tags and items put by the environment. wait returns when the graph program completes.

  - When wait returns control to the environment, the environment can perform gets.
- The environment also has access to an iterator to get all the instances in a collection without knowing their tags. This is facility is not available to steps.
- For the graph to execute some step and produce some result, the environment must put at least one tag and get at least one tag or item.

1.2.6 Semantics

As the program executes, instances of the three types of objects monotonically accumulate attributes indicating their state. The set of attributes for instances of the tags, items and steps and the partial ordering in which an instance can acquire these attributes is shown in

Figure 1-2.

- When (S: t1) executes, if it produces [I: t2], then [I: t2] becomes *available.*
- When (S: t1) executes, if it produces <T: t2>, then <T: t2> becomes *available.*
- If <T> prescribes (S), when <T: t> is *available* and then (S: t) becomes *prescribed.*
- If forall [I, t1] such that (S: t2) gets [I, t1]

[I, t1] is *available* // if all inputs of (S: t2) are available

then (S: t2) is *inputs-available.*

- If (S: t) is both *inputs-available* and *prescribed* then it is *enabled.*

Any *enabled* step is ready to execute.

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

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

Program termination occurs when no step is currently executing and no unexecuted step is currently *enabled*.

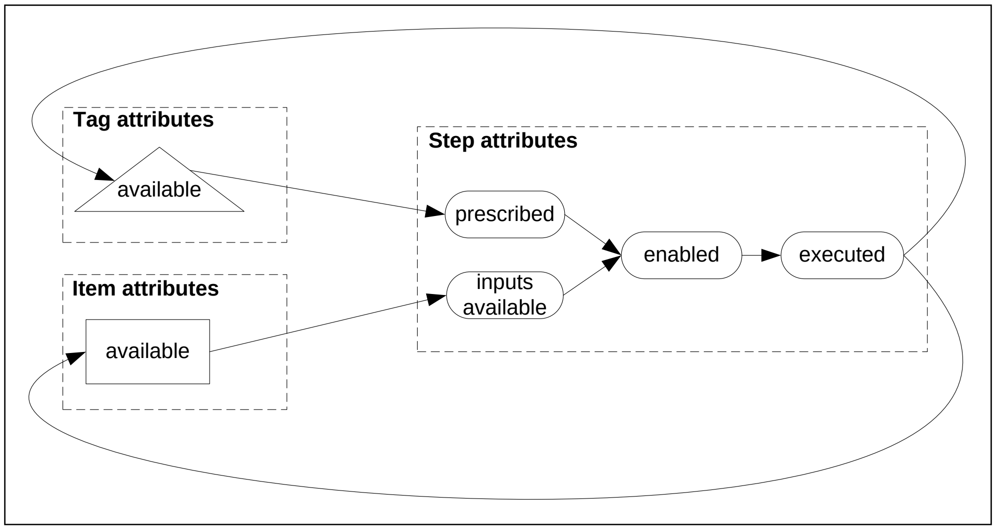


Figure 1-2 Semantics: attributes of instances

Valid program termination occurs when a program terminates and all prescribed steps have executed.

The program produces the same results regardless of the schedule within or the distribution among processors. It is possible to write an invalid program, one that stops with steps that are prescribed but whose input is not available. However, altering the schedule or distribution will not change this result.

Note that the semantics allow but do not require parallel execution. This makes it easier to develop and debug an application on a uniprocessor.

1.2.7 Summary

Intel® Concurrent Collections for C/C++ is a way of expressing a program:

IN TERMS OF HIGHER-LEVEL OPERATORS AND DATA STRUCTURES APPROPRIATE TO THE APPLICATION.

This allows the programmer to continue to express the finest grain aspects of the program in any familiar serial programming language. Only the relationships among the higher-level operators are expressed in this language.

IN DYNAMIC SINGLE ASSIGNMENT FORM.

This means that the computation is expressed in terms of values, not locations. Each high-level operation is functional. The only side-effects are explicitly producing values. There is no overwriting so there are no race conditions.

IN TERMS OF ORDERING CONSTRAINTS BASED ON THE FLOW OF DATA AND CONTROL.

The operations of the computation are partially ordered based only on the data flow among them. There is no need for analysis to undo the serial ordering.

This model delivers to the domain expert computation that

- is based on how people actually communicate with one another about their application,
- is race free,
- gets identical results regardless of the schedule, distribution, configuration or architecture
- requires no reasoning about parallelism and
- is neutral with respect to target platform.

and delivers to the tuning expert (person or program)

- maximal flexibility for tuning. This flexibility comes about because only the constraints are explicit. There is no overwriting or arbitrary serialization to complicate scheduling and distribution decisions. We expect improved performance based on increased flexibility in scheduling and distribution.

2 The Eight design patterns

Figure 2-1 the four possible relations between steps. Notice that any two steps may have the following relationships:

- one produces data the other uses or not (column A or B)
- one computes the control for another or not (row 1 or 2)
- the two steps are in distinct collections or in the same collection (table I or II)

The two possibilities for each of these three characteristics result in the eight distinct design patterns shown in the figure.

Notice that the patterns in this figure are not complete programs, they are simply fragments.

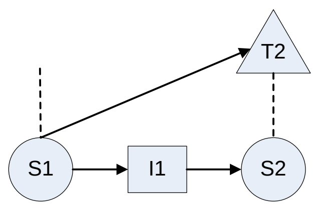
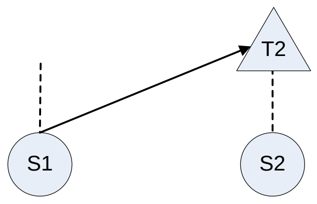
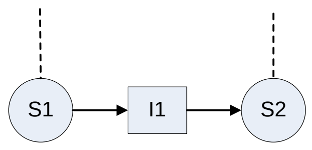
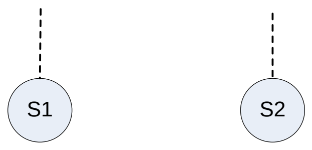
I: **Distinct collections**	A: Producer/Consumer	B: No Producer/Consumer
1: Controller/Controllee		
2: No controller/Controllee		

Figure 2-1 the four possible relations between steps in distinct collections

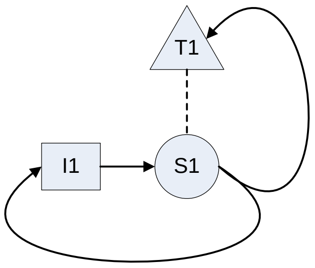
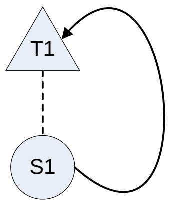
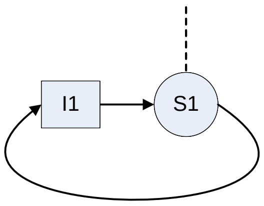
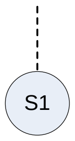
II: **Same collection**	A: Producer/Consumer	B: No Producer/Consumer
1: Controller/Controllee		
2: No Controller/Controllee		

Figure 2-2 the four possible relations between steps in the same collection

The next eight sections discuss the eight primitive design patterns from Figure 2-1 the four possible relations between steps With each pattern we provide an example or two. These examples are very tiny. They are unrealistic in the size of both data and the computation. Real applications would have more complex graphs, the size of the item instances would be much larger and the number of executed instructions within each step instance would be much larger. The intent here is to provide some simple working examples to show the basics.

These eight patterns describe how two steps can relate to each other *directly*. There are clearly *indirect* analogs of these relations but they are not discussed here.

With each example we provide

- A set of figures showing

  - the graphical representation of the example

  - the textual representation of the example

  - the serial step codes for the example

  - the environment code for the example

- Several brief discussions including

- how the example relates to the pattern

- a discussion of the example

- the constraints on parallelism within the example and

- a few suggestions of modifications of the example to try

The order in which the patterns are presented is a reasonable one and you might want to read this document from front to back. If you prefer, feel free to start exploring any pattern you choose. We suggest that you read one that involves steps from the same collection, one that involves steps from distinct collections, one that has a producer / consumer relation and one that has a controller/controlee relation.

3 Producer/consumer (distinct step collections)

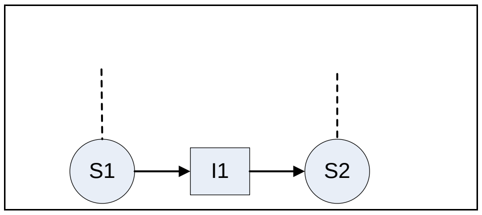


Figure 3-1 pattern: between collections; producer/consumer; no controller/controllee

Since every step collection is prescribed by a tag collection, there is a dotted line from each step. If a tag collection is not produced within the pattern, the specific tag collection is not indicated. In this pattern, the two step collections might be prescribed by the same or distinct tag collections. The pattern allows for both possibilities.

3.1 Example: producer consumer

3.1.1 Graph specification

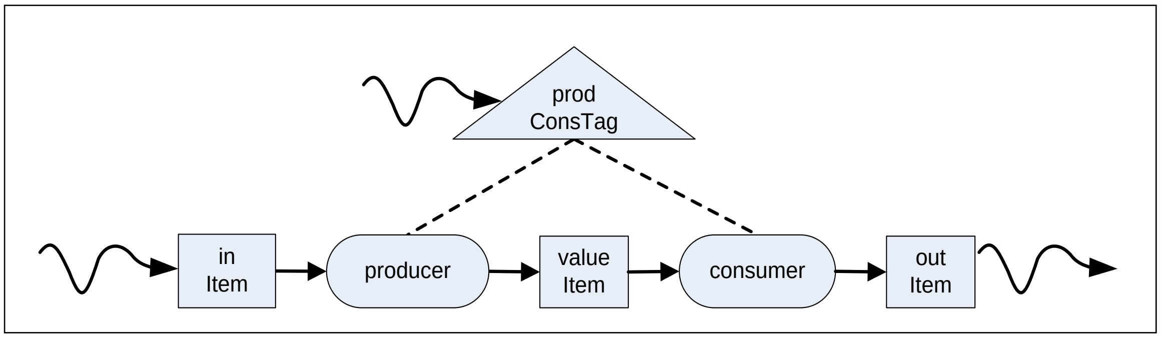


Figure 3-2 example: producer consumer

```
[int inItem<int>: int tag];      // items consumed by producer
[int valueItem<int>: int tag];  // items produced by producer
// and consumed by consumer
[int outItem<int>: int tag];     // items produced by consumer
<int prodConsTag: int tag>;     // tags prescribing
```

```
// producer and consumer steps
(producer);
(consumer);

// For each instance of prodConsTag there will be an instance of
// producer and an instance of consumer
<prodConsTag> :: (producer);
<prodConsTag> :: (consumer);
// environment produces all inItems and prodConsTags
env -> [inItem], <prodConsTag>;

// Producer consumes inItem and produces valueItem
[inItem: tag] -> (producer: tag) -> [valueItem: tag];

// Consumer consumes valueItem and produces outItem
[valueItem: tag] -> (consumer: tag) -> [outItem: tag];

// outItems are consumed by the environment
[outItem] -> env;
```

3.1.2 The Program (C++)

An example implementation of this pattern and specification can be found in the samples directory: samples/producer_consumer.

3.1.3 How pattern and example relate

(producer) in this example, corresponds to (S1) in the pattern. (consumer) in this example, corresponds to (S2) in the pattern and [valueItem] corresponds to [I1] in the pattern.

3.1.4 Discussion

This example is based on two steps collections (producer) and (consumer). They are both prescribed by the same tag collection <prodConsTag> so for each tag in that collection, there will be an instance of a producer and an instance of a consumer. The example includes input to (producer) and output from (consumer). These collections, [inItem] and [outItem] are produced and consumed by the environment respectively. In the step code, the producer multiplies the contents of the [inItem] by 10 and puts that value as [valueItem]. The consumer gets the [valueItem] and adds 3 to it before putting it out as [outItem].

3.1.5 Parallelism

The instance of (consumer) that consumes an item instance j can not executed before the instance j is produced by the instance of the (producer). But different producer/consumer pairs are independent of each other and can execute in any ordering with respect to each other.

4 Controller/controller (distinct step collections)

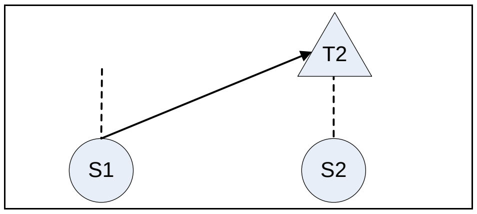


Figure 4-1 pattern: between collections; no producer/consumer; controller/controllee

4.1 Example: generate iterations

4.1.1 Graph specification

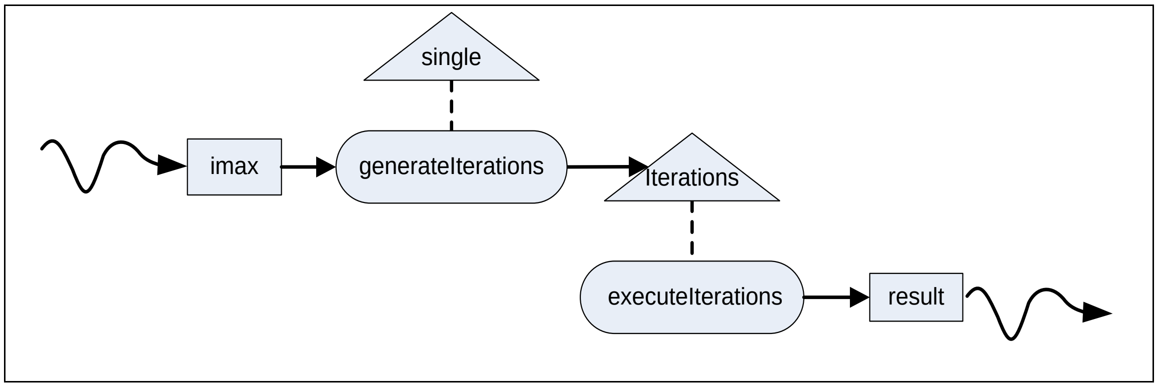


Figure 4-2 example: generate iterations

```
// declarations
<int single: int s>;              // single tag value
[int imax<int>: int s];           // single value of imax
<int iterations: int i>; // contains tags 1 to imax
[myType result<int>: int i];      // result produced by iteration i

// inputs
```

```
env -> <single>;
env -> [imax];

// prescriptions
<single> :: (generateIterations);
<iterations> :: (executeIterations);

// producer/consumer relations
[imax] -> (generateIterations);
(generateIterations) -> <iterations>;
(executeIterations) -> [result];

// outputs
[result] -> env;
```

4.1.2 The Program (C++)

An example implementation of this pattern and specification can be found in the samples directory: samples/gen_new_iter_space.

4.1.1 How pattern and example relate

(generateIterations) in this example, corresponds to (S1) in the pattern. (executeIterations) in this example, corresponds to (S2) in the pattern and <iterations> corresponds to <t2> in the pattern.

4.1.2 Discussion

The (generateIterations) step collection contains only a single instance determined by the <single> tag collection. (generateIterations) generates the <iterations> tag collection. This tag collection contains one tag instance for each iteration from 1 to imax. This <iterations> tag collection controls the (executeIterations) step collection. So (executeIterations) will execute once for each iteration. When each instance executes it may produce a [result] items.

One might ask why the (generateIterations) collection which only has a single instance is not simply performed in the serial environment. The tag collection <iterations> could just be input to the graph.There are several answers to this question. It could well be part of the environment. That would be fine. On the other hand if this fragment were part of a larger program, this approach might make more sense for other reasons. For example, if some significant part of the computation results in the value of imax or input to (executeIterations), there is no need to go out to the environment and back again. The environment can contain serial code between parallel regions. As another example, if the tags were modified to include another tag component, then instead of a serial computation generating a one-dimensional computation, we would have a one-dimensional computation generating a two-dimensional computation. This approach would make more sense.

4.1.3 Parallelism

Each instance of (executeIterations) must wait until it is prescribed by a tag in the <iterations> collection. This means they can only execute after the single instance of (generateIterations) is complete. But there are no ordering constraints among the instances of (executeIterations).

4.1.4 Graph specification

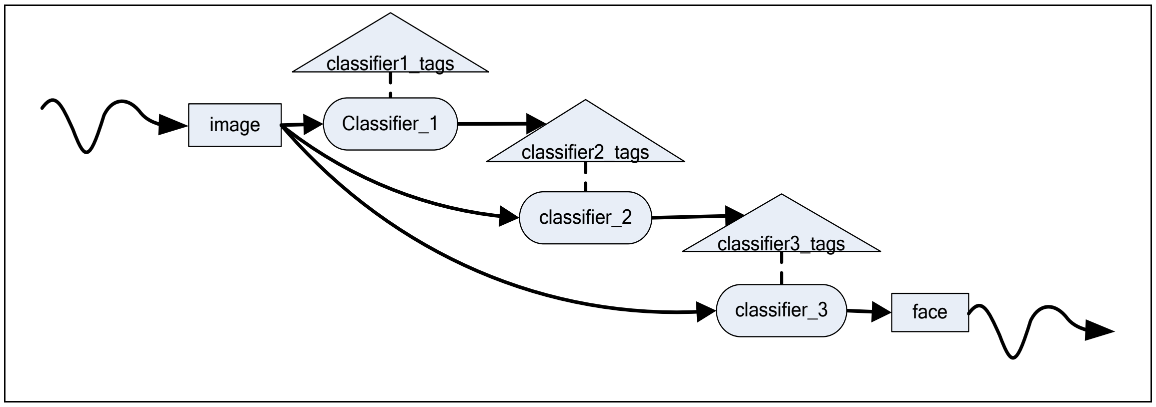


Figure 4-3 example: face detector

```
// Declarations
[myImageType image <int>: imageID];   // Input images
[myImageType face <int>: imageID];    // Images recognized as a face
<int classifier1_tags: imageID>;      // First classifier tag collection
<int classifier2_tags: imageID>;      // Second classifier tag collection
<int classifier3_tags: imageID>;      // Third classifier tag collection

// Step Prescriptions
<classifier1_tags> :: (classifier1);
<classifier2_tags> :: (classifier2);
<classifier3_tags> :: (classifier3);

// Input from the caller; a set of images and a set of
// tags that assign the images to be processed by classifier1
env -> [image], <classifier1_tags>;

// Step Executions
[image] -> (classifier1) -> <classifier2_tags>;
[image] -> (classifier2) -> <classifier3_tags>;
```

```
[image] -> (classifier3) -> [face];

// Return the faces to the caller
[face] -> env;// Declarations
```

4.1.5 The Program (C++)

An example implementation of this pattern and specification can be found in the samples directory: samples/FaceDetection.

4.1.6 How pattern and example relate

There are two instances of the pattern in this example. (classifier1), <classifier2_tags> and (classifier2) correspond to (S1), <T2> and (S2) in the pattern. In addition, (classifier2), <classifier3_tags> and (classifier3) in the example also correspond to (S1), <T2> and (S2) in the pattern.

4.1.7 Discussion

This is a severe abstraction of a cascade face detector used in the computer vision community. Instead of using vision techniques on an image we use text with words such as "nose" and "mouth". However the form of the graph specification is real. It includes a cascade of classifiers. The first classifier, called (classifier1) operates on images. It is controlled by a tag collection, called <classifier1_tags>. This tag collection contains tags specifying the image. If the first classifier determines that an image might be a face, it produces that same tag value but into a different tag collection, <classifier2_tags> that controls the second classifier. This tag collection will be a subset of the first tag collection. An image may fail on any classifier but if it makes its way to the end, it is deemed to be a face. If an image passes the last of the classifiers, it is identified in the item collection [face].

In the real cascade face detector, the classifiers are determined by machine learning, but you might think of them as looking for eyes, nose, mouth, etc. Also the real face detector examines not only whole images but many overlapping subimages of the same image.

4.1.8 Parallelism

The constraints on parallelism for this application are that an instance of (classifier2) can not execute until the associated instance of (classifier1) has executed and determined that it might be a face. Similarly an instance of (classifier3) has a control dependence on (classifier2). Instances of a given classifier can execute at the same time and instances of distinct classifiers can execute at the same time as long as the earlier constraint is met.

5 Both (distinct step collections)

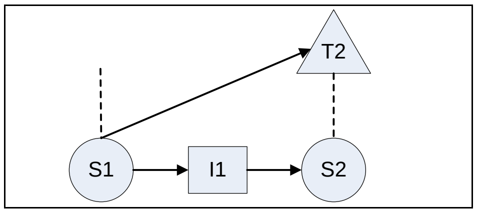


Figure 5-1 pattern: between collections; producer/consumer; controller/controllee

5.1 Example: partition string

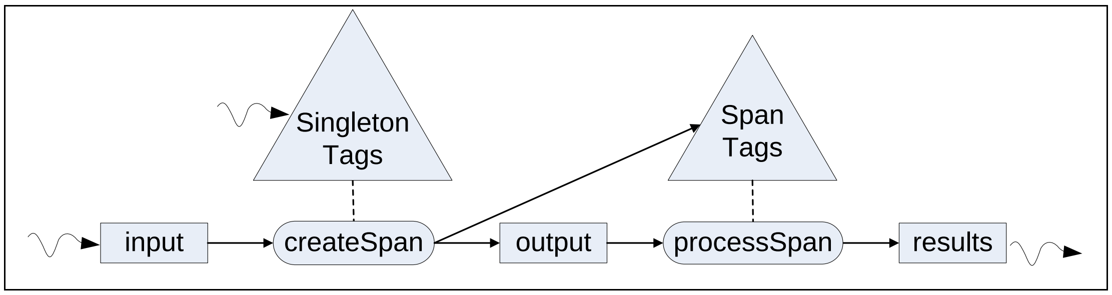


Figure 5-2 example: partition string

5.1.1 Graph Specification

```
// declarations

<int singletonTag: singleton>;
<int spanTags: spanID>;
[string input <int>: singleton];
[string span <int>: spanID];
[string results <int>: spanID];

<singletonTag> :: (createSpan);
<spanTags> :: (processSpan);

// program inputs and outputs
```

```
env -> [input];
env -> <singletonTag>;
[results] -> env;

// producer/consumer relations

[input: singleton] -> (createSpan: singleton);
(createSpan: singleton) -> <spanTags: spanID>, [span: spanID];
[span] -> (processSpan) -> [results];
```

5.1.2 The Program (C++)

An example implementation of this pattern and specification can be found in the samples directory: samples/partition_string.

5.1.3 How pattern and example relate

(createSpan) in this example, corresponds to (S1) in the pattern. (processSpan) in this example, corresponds to (S2) in the pattern and <spanTags> corresponds to <t2> in the pattern.

5.1.4 Discussion

This program inputs a string and generated a collection of strings that partitions the input into subparts that contain the same character

e.g., If the single instance of [input] contains "aaaffqqqmmmmmmm"

then (createSpan) creates the following instances of [output]

"aaa"

"ff"

"qqq"

"mmmmmmm"

these instances are processed by (processSpan).

In our example:

The type of items in [input] is a single string of char. [span] is a collection of strings of char. <singletonTag> a single tag instance used for createSpan. <spanTags> identifies the instances in the span collection. (createSpan) creates that span from the input. (processSpan) processes the created span.

One might ask why we execute the (createSpan) step, which only happens once, within the graph and not in the environment. We could do this. If this fragment were part of a larger program then in some situations it might be better not to.

- For example, if there were a collection of strings as input to be processed, there would be another tag component specifying the string.
- Another example might be that the string to be processed is the output of significant earlier part of the program and it also had potential parallelism. There is no need to go out to the environment for the serial work and then back to a graph program for each transition between serial and parallel parts of the application.

5.1.5 Parallelism

Consider the potential parallelism in this example. The single instance of (createSpan) must occur first since it generates the [output] items to be processed. Then the (processSpan) steps can execute in parallel.

6 Neither (distinct step collections)

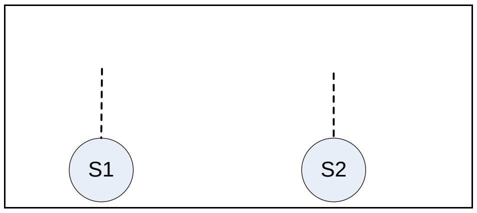


Figure 6-1 pattern: between collections; no producer/consumer; no controller/controllee

We have not included an example for this case. Basically the two step collections have no direct relation to each other. (s1) does not produce any tags that prescribe (s2). (s1) does not produce any data consumed by (s2).

7 Producer/consumer (same step collection)

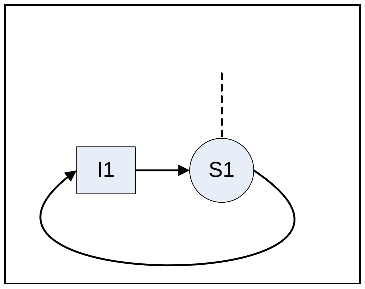


Figure 7-1 pattern: within a collection; producer/consumer; no controller/controllee

7.1 Example: reduce rows

7.1.1 Graph specification

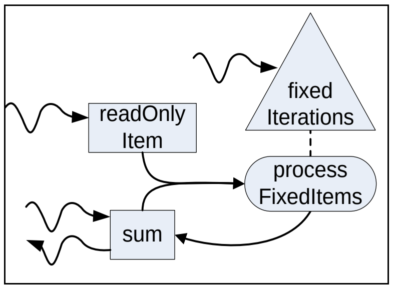


Figure 7-2 example: reduce rows

```
// declarations
<pair fixedIterations>;      // tag: (row, col)
[int readOnlyItem <pair>];   // item tag: (row, col)
[int sum <pair>];            // item tag: (row, col)

<fixedIterations> :: (processFixedItems);
```

```
// program inputs and outputs
env -> <fixedIterations>;
env -> [readOnlyItem];
env -> [sum];     // initialized to contain the value zero
[sum] -> env;

// producer/consumer relations
[readOnlyItem: row, col],
[sum: row, prev(col)]
         -> (processFixedItems: row, col);

(processFixedItems) -> [sum];
```

7.1.2 The Program (C++)

An example implementation of this pattern and specification can be found in the samples directory: samples/reduce_rows.

7.1.3 How pattern and example relate

(processFixedItems) in this example, corresponds to (S1) in the pattern. [sum] in this example, corresponds to [I1] in the pattern.

7.1.4 Discussion

This program operates on a two dimensional collection tagged by row and col. The environment produces a collection of items together with a corresponding collection of tags. It also produces an intial value of the [sum]. Within each row, the program processes the input items in order across the columns and computes a running sum into [sum].

7.1.5 Parallelism

The constraints on parallelism in this example are that the columns in a given row must be processed in order. The processing between different rows is totally independent.

8 Controller/controllee (same step collection)

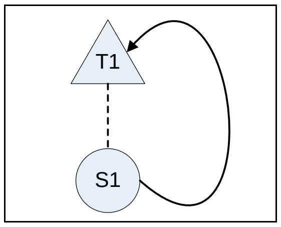


Figure 8-1 pattern: within a collection; no producer/consumer; controller/controllee

8.1 Example: search in sorted tree

8.1.1 Graph specification

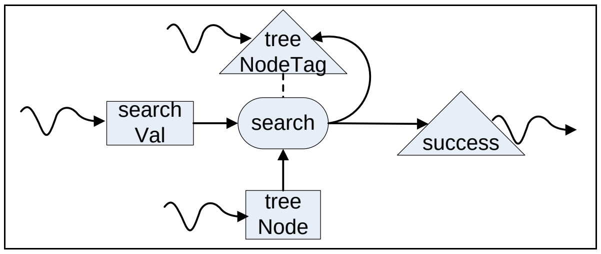


Figure 8-2 example: search in sorted tree

```
// declarations
<pair searchTag>;
[int searchVal <int>];
[myNodeStruct treeNode <int>];
<pair success>; // identifier of the node where the value was found.
                // if the value was not found, no tag will be produced.
<searchTag> :: (search);

// program inputs and outputs
env -> <searchTag>;
env -> [searchVal];
env -> [treeNode];
<success> -> env;
```

```
// producer/consumer relations
[searchVal], [treeNode] -> (search);
(search) -> <searchTag>, <success>;
```

8.1.2 The Program (C++)

An example implementation of this pattern and specification can be found in the samples directory: samples/search_in_sorted_tree.

8.1.3 How pattern and example relate

(search) in this example, corresponds to (S1) in the pattern. <treeNodeTag> corresponds to <T1> in the pattern.

8.1.4 Discussion

This program searches a sorted tree for a specific integer. The tree contains sorted ints. The ints themselves are at the leaves. The intermediate nodes contain the largest int in the left (smallest) subtree. This value is called the partition value. This value is used to direct the search down either the right or left child.

The tree is input as a collection of items together with a corresponding collection of tags. Another input item is a single item containing an int we are searching for. The process starts at the top of the tree and at each intermediate node processes the right or left branch of the tree depending on the partition value.

The tree nodes are identified by an int containing only zeros and ones. The root is identified as one. Zero indicates the left child, one indicates the right child. Node 1011 is the right child of the right child of the left child of the root. This mechanism is used for now because the initial implementation only supports integer tags.

8.1.5 Parallelism

Within a search for a given [searchVal] the processing down the tree is serial. It starts at the root and continues down the tree following right or left children in order. However, processing of between distinct instances of [searchVal] is totally independent.

9 Both (same step collection)

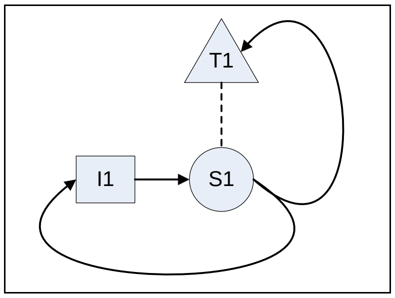


Figure 9-1 pattern: within a collection; producer/consumer; controller/controllee

9.1 Example: reduce Rows until predicate

9.1.1 Graph specification

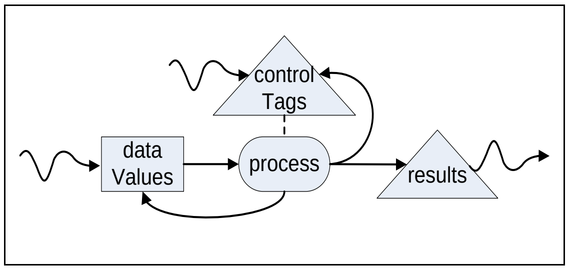


Figure 9-2 example: reduce rows until predicate

```
// declarations

[myType dataValues <pair>];      // item tag: (inputID, current)
[int results <int>];             // item tag: inputID
<pair controlTags>;              // tag: (inputID, current)

<controlTags> :: (process);

// program inputs and outputs

env -> [dataValues], <controlTags>;
[results] -> env;
```

```
// producer/consumer relations

[dataValues] -> (process);

(process: inputID, current)
    -> [dataValues: inputID, inc(current)],  // this is the data
dependence
       <controlTags: inputID, inc(current)>, // this is the control
dependence
       [results: inputID];
 // inc(current) increments current by 1
```

9.1.2 The Program (C++)

An example implementation of this pattern and specification can be found in the samples directory: samples/reduce_rows_until_pred.

9.1.3 How pattern and example relate

(process) in this example, corresponds to (S1) in the pattern. [dataValues] in this example, corresponds to [I1] in the pattern and <controlTags> corresponds to <T1> in the pattern.

9.1.4 Discussion

This program inputs a set of input items tagged by inputID. For each input, it applies a function f to create a new value for inputID. The different values computed for the same inputID are distinguished by a second tag component called current. When predicate p is true of the value, the processing for that inputID is complete. The value when predicate p is true is stored in [result]. [result] only distinguishes among distinct inputID values. The iteration on which the predicate became true is no longer needed. [result] is emitted to the environment.

The [dataValues] contains a struct that contains the two fields, the latest value, the running sum

Function f uses the current value as a seed to random number generator to produce the next value.

The predicate p succeeds when the sum is greater than some number. [results] indicates the value that passed the predicate test.

9.1.5 Parallelism

The one constraint on parallelism is that the running sum for each inputID must be processed in order. There are no ordering requirements among the different inputIDs. They can be processed in parallel.

9.2 Example: divide and conquer

9.2.1 Graph specification

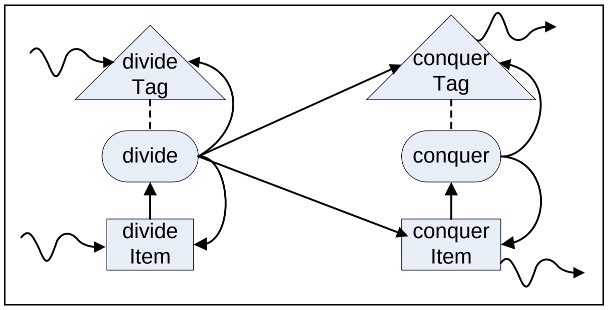


Figure 9-3 example: divide and conquer

```
// Declarations
[int divideItem <int>];
[int conquerItem <int>];
<int divideTag>;
<int conquerTag>;

// Step Prescriptions
<divideTag> :: (divide);
<conquerTag> :: (conquer);

// Input from the caller: a root item and a root tag
env -> [divideItem: rootTag], <divideTag: rootTag>;

// Step Executions
[divideItem: tag] -> (divide: tag);
(divide: tag)    -> [divideItem: leftChild(tag)],
                    [divideItem: rightChild(tag)],
                    <divideTag: leftChild(tag)>,
                    <divideTag: rightChild(tag)>,
                    [conquerItem: tag],
                    <conquerTag: parent(tag)>;
```

```
[conquerItem: leftChild(tag)],
     [conquerItem: rightChild(tag)] -> (conquer: tag);
(conquer: tag) -> [conquerItem: tag], <conquerTag: parent(tag)>;
// Return to the caller
[conquerItem: rootTag] -> env;
```

9.2.2 The Program (C++)

An example implementation of this pattern and specification can be found in the samples directory: samples/divide_and_conquer.

9.2.3 How pattern and example relate

This example contains two instances of the pattern.

(divide) in this example, corresponds to (S1) in the pattern. [divideItem] in this example, corresponds to [I1] in the pattern and <divideTag> corresponds to <T1> in the pattern.

In addition, (conquer) in this example, corresponds to (S1) in the pattern. [conquerItem] in this example, corresponds to [I1] in the pattern and <conquerTag> corresponds to <T1> in the pattern.

9.2.4 Discussion

In the example, the [divideItem] and [conquerItem] each contain a single integer. (divide) puts out two [dividItem] children. The contents of the two children add up to the contents of the parent. (conquer) simply adds these values.

9.2.5 Parallelism

Each (divide) step requires that it parent (divide) has already executed and produced the childs' [divideItem] but two (divide) steps such that neither is an ancestor of the other can execute independently. Each (conquer) step required that both children [conquerItems] are available before the parent (conquer) executes. In addition, (divide) steps and (conquer) steps are independent if there is no leaf step that is a decendent of both the (divide) step and the (conquer) step.

10 Neither (same step collection)

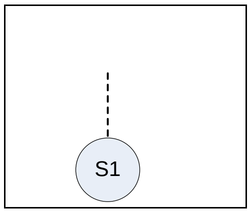


Figure 10-1 pattern: within a collection; no producer/consumer; no controller/controllee

10.1 Example: primes

10.1.1 Graph specification

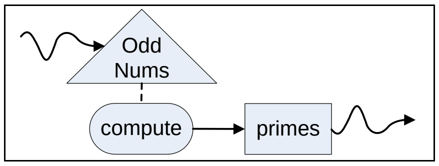


Figure 10-2 example: primes

```
// Declarations
// The tag values are the odd numbers in the range [3..n]
<int oddNums>;
// The prime numbers as identified by the compute step
[int primes <int>];

// Step prescription
// For each oddNums instance, there is
<oddNums> ::  (compute);

// Step execution
// The compute step may produce a prime number
// (in the form of a item instance)
(compute) -> [primes];

// Input from the environment: initialize all tags
env -> <oddNums>;
```

```
// Output to the environment is the collection of the prime numbers
[primes] -> env;
```

10.1.2 The Program (C++)

An example implementation of this pattern and specification can be found in the samples directory: samples/primes.

10.1.3 How pattern and example relate

(compute) in this example, corresponds to (S1) in the pattern.

10.1.4 Discussion

In this example there is a single step collection called (compute). It is prescribed by the tag collections <oddNums>. This means that for each tag in <oddNums> there will be a corresponding (compute) step with access to that tag value. The (compute) step will determine if the tag value is a prime number. Notice that the (compute) step does not consume any input items. It only operates on its tag value. The output of this program is the [primes] item collection, one instance for each input that was a prime.

10.1.5 Parallelism

Consider the potential parallelism. An instance in (compute) can execute as soon as its tag is available. There are no constraints between one instance of (compute) and another. The instances may all execute in parallel or in any order.

11 SUMMARY

This document introduces Intel® Concurrent Collections for C++. A major goal of this model is a separation of concerns of the domain-expert and the tuning-expert. This document is focused on the needs of the domain expert and shows how to express the semantics of the program without any concern for parallelism. It presents the basic concepts of the graph specification. and the semantics of the language.

A major focus is the eight patterns that identify the eight ways that any two steps might relate to each other. For each pattern we have provided real but very tiny examples. These are intended to provide an understanding of how to think in this model and how a program written in the model is put together. Implementations of the examples are available with the distribution of Intel® Concurrent Collections for C++.