A Study of Computational Reconfiguration in a Process Network

D.L. Webb, A.L. Wendelborn and J. Vayssière

Department of Computer Science, University of Adelaide
South Australia 5005, Australia

Email: {darren, andrew, julien}@cs.adelaide.edu.au

Abstract

In the process network model, the network evolves by reconfiguration. Reconfiguration changes the representation of the network. With a multi-threaded implementation of the process network system, it is necessary to coordinate concurrent accesses to the representational structure. We compare two approaches to the problem of ensuring consistency of representation during reconfiguration. One is a "localized" view, taken from the viewpoint of the process undergoing reconfiguration. The other requires a "global" view of the entire process network structure. We show how reconfiguration takes place in each case, and compare advantages and disadvantages of each.

1 A Technique for Process Network Reconfiguration

The Kahn model of process networks [Kah74] is used to represent transformations to streams of data. Process networks are structured as directed graphs where a node represents a process and an edge represents the flow of data from one process to another. Kahn defines a process to be a mapping from one or more input streams (or histories) to one or more output streams. Processes communicate only through directed first in - first out (FIFO) stream of tokens with unbounded capacity such that each token is produced exactly once, and consumed exactly once. Production of tokens is non-blocking, while consumption from an empty stream is blocking. The model is highly concurrent and deterministic, and is of interest to us as a semantically sound formulation of flow-based systems.

Reconfiguration is a fundamental element of process network semantics as defined by [Kah74], via the recursive process schema. In fact, a process network can be regarded as starting with one node and expanding as the computation proceeds [KM77]. The behaviour of process networks is dynamic, evolving in a top-down fashion. Process networks reconfigure by replacing a node with a subgraph. This is only possible if the subgraph can be appropriately spliced into the incoming and outgoing edges of the original node. Here, we examine aspects of safe implementation of such computational reconfiguration (mutation of a representation), as a basis for further work on adaptive reconfiguration.

[Kah74] assumes normal order (demand driven) evaluation, hence:

1. reconfiguration happens only when it is needed, and
2. no unnecessary reconfigurations occur.

If an implementation is also demand driven, then (1) makes it possible to make strong statements about the process network node that is undergoing reconfiguration, and its input and output channels.

Implementing reconfiguration requires that we change the process network representation dynamically; we must be able to guarantee that the mechanics of mutating the process network's implementation structure gives the correctly formed process network before and after mutation, and that mutation must ensure that all channel connections remain correct, that no values are lost from channels, that no values are introduced or duplicated, and that the process network correctly resumes operation. We refer to this as ensuring consistency of representation during reconfiguration.

Again using (1) above, and a demand driven implementation, we can state (in Fig. 1) that the (single) output channel of B is empty and hungry, and that a value must be produced on that channel immediately after the reconfiguration in order to satisfy that demand; in [Wen82], we showed how to perform the mutation safely under these conditions in both quasi-parallel and distributed process network implementation schemes.

But we do not want to restrict operation of a process network to just demand driven operation; it restricts concurrency (not entirely, as a sink node can propagate demands on any number of inputs, thereby providing several concurrent activities to satisfy them, but it precludes pipelining or streaming parallelism). For this reason, we want parts of a network to proceed in a data driven fashion. This means we need to take more care in handling mutations.

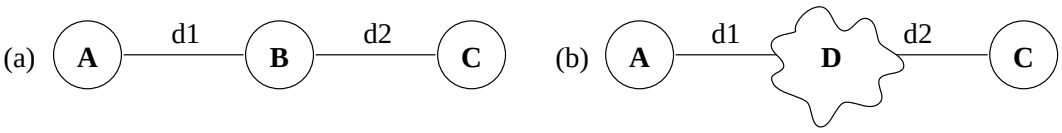
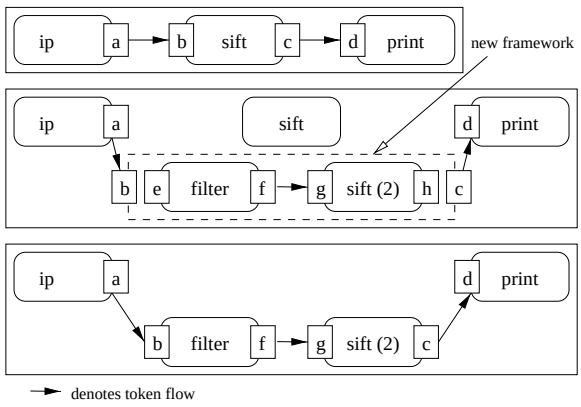


Figure 1: Expansion of a network from (a) to (b), replacing the existing node *B* with a subgraph *D*. *B* may or may not survive as part of *D* in the new configuration.




```
// create a new framework
Builder builder = Toolkit.newBuilder(sift1);

// construct the new subgraph
Filter filter = new Filter(prime);
Sift sift2 = new Sift();
builder.add(filter);
builder.add(sift2);
OutputPort f = filter.getOutputPort(0);
InputPort g = s.getInputPort(0);
builder.connect(f,g);

// splice framework into existing network
InputPort b = builder.getInputPort(0);
InputPort c = builder.getOutputPort(0);
filter.setInputPort(b,0);
sift2.setOutputPort(c,0);

// schedule any new nodes
builder.trigger();
```

Figure 2: Code for the expansion of a process network.

We cannot guarantee that the output channel will be empty, or that its consumer will be blocked; also, producers of its input channels may be active.

Specifically, we must be able to guarantee (in Fig. 1) that the transformation from *B* to *D* is implemented by a safe mutation process. We must recognize with data driven evaluation that *A* and *C* are autonomous so potentially will produce to d_1 and consume from d_2 while reconfiguration occurs. For this reason, the endpoints of the channels held by *A* and *C* cannot be modified. This leaves *B* and the nodes in *D*, the nodes affected in the reconfiguration, as the remaining points for contention.

During the process of reconfiguration, it is likely that the endpoints of *B* and *D* will change. To ensure consistency of representation, we assume that once the mutation starts, it continues to completion. This assumption ensures there is no activity involving the mutating process, hence no opportunity to adversely affect the consistency of the existing or newly created intermediary channels in *D*.

Our technique for ensuring consistency is to create an entirely separate "framework" in which the new subgraph *D* is created, seamlessly splicing the input and output ports of *B* with those of *D*, and starting any newly created process network nodes. From the time that *B* decides to reconfigure until newly formed nodes in *D* are started, neither *B* nor the nodes of *D* are able to manipulate any channels and hence consistency is preserved.

To illustrate our technique (described in detail in [VWW99]), we look at the example of the Sieve of Eratosthenes. This example is instructive because the computation evolves by reconfiguration, and forms a pipeline of filter and sieve processes in so doing. Further, it can be written either recursively (a sieve is replaced by a filter and a new sieve at the discovery of each prime) or iteratively (a filter is inserted in front of the sieve process which remains in the process network). In Fig. 2, we show the ports of the reconfiguring process (*b* and *c*) assigned to the framework, the new processes (`filter1` and `sift2`) created within the framework, as well as their ports (*e*, *f*, *g* and *h*), and the internal connection of *f* to *g*. The mutation is completed by distinguishing appropriate ports of newly created processes within the framework, and identifying them with the ports of the reconfiguring process (here, *b* becomes *e* and *c* becomes *h*). We schedule the newly formed processes using `trigger()`.

Process network nodes can also reconfigure by disappearing from the process network; [KM77] describe this as "tieing" the input of a process network to its output. This type of reconfiguration is especially useful in defining cyclic

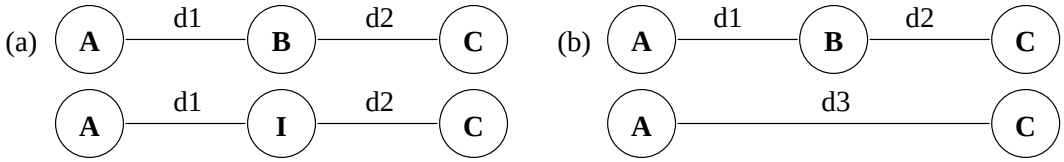
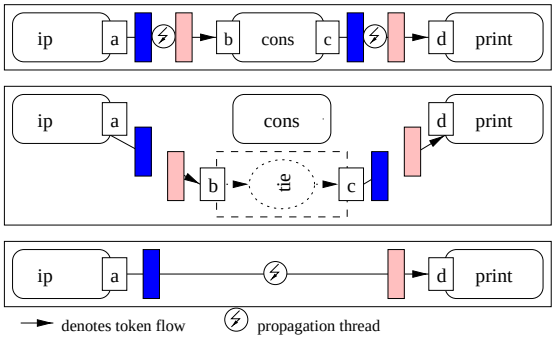


Figure 3: Shrink a network. (a) provides quasi-shrinking, whereas (b) actually shrinks the network.




```
// create a new framework
Builder builder = Toolkit.newBuilder(cons);

// tie the framework's ports
InputPort b = builder.getInputPort(0);
OutputPort c = builder.getOutputPort(0);
builder.tie(b,c);
```

Figure 4: Disappearing a `cons` operator

networks. Just drawing a cycle is meaningless as a computation because it implies immediate deadlock. But if we use a process which inserts $d(\geq 1)$ values into a channel, and then disappears, we have effectively specified a "delay" d in the cycle and turned in into a feedback loop – V_i is computed in terms of $V_{i-d}, \ldots, V_{i-i}$.

Figure 3(a) shows how such a tie may be implemented using expansion. We can expand the network to replace B with the identity node I. In other words, B emits its prefix, then behaves as an identity process. But we should avoid the overhead of copying tokens unchanged. Hence, for disappearing to be efficient and useful, the "disappear" mutation must transparently combine two channels into one, without leaving artifacts (such as hidden redirection of I). It is the operation of combining channels that makes this type of reconfiguration different from, and more difficult than, that of Fig. 1 ("expansion").

In [Wen82], under the assumption of demand driven evaluation, we could state that d_2 must be empty, so $d_1 + d_2 \rightarrow d_1$. We cannot apply the same result for data driven evaluation as it is possible that d_2 is not empty. In this case, we must merge the histories such that $d_1 + d_2 \rightarrow d_3$. This leaves us with the problem of preserving the ordering of d_1, d_2 and the resulting d_3 during mutation.

Under the consistency rules for expansion, we can not modify the endpoints at A and C. Once again, the autonomous nature of these nodes complicates the mutation operation, as it is possible for d_1 and d_2 to be modified (possibly through another disappear operation) by A and C prior to establishing d_3. It is undesirable to require the involvement of A and C in the mutation operation (i.e. stopping them), hence we look at the channel architecture to assist performing this mutation.

Our technique for channel merging requires that we describe the channel architecture in two halves, d^p on the producer side and d^c on the consumer side, with a thread coordinating propagation of tokens from d^p to d^c. We refer to this architecture as *half-channels*. Half-channels enable us to insulate A and C from any manipulation of their endpoints by requiring that d_1^p and d_2^c remain after the mutation. The effect of the mutation is then a process of gaining a mutation lock on all four half-channels, stopping the propagating threads, propagating any tokens left in d_1^c and d_2^p to d_2^c, creating and starting a new propagating thread between d_1^p and d_2^c, then releasing the remaining channel mutation locks.

To illustrate this channel architecture (described in detail in [VWW]) we consider the implementation of the `cons` operator. The `cons` operator prefixes a token to a channel and then vanishes. Figure 4 shows the creation of a new framework to replace the existing node, but on this occasion we take the two ports of the new reconfiguration and tie them. The `tie()` operation stops the propagating threads of the two channels, propagates any tokens in the half-channels that disappear with `cons`, then create and start a new propagating thread to resume the progress of the channel.

With that, we have a mutation scheme that:

- is safe under both demand driven and data driven evaluation;

- involves only the reconfiguring process and its input and output channels;
- is amenable to distributed implementation.

There is another aspect of data driven evaluation we should consider. It is speculative by nature in that it computes values that may not be needed to produce the overall process network result. In particular, it may cause initiation of non-terminating computation in a fragment of the process network, but if the results of that computation are not actually used downstream, the process network will still function correctly (we assume but don't discuss a suitable throttling or resource monitoring mechanism to suppress the errant fragment).

The point here is that, in considering mutation for reconfiguration in a separate part of the process network, we should not let the presence of this non-terminating fragment delay the mutation: this makes it essential that the mutation mechanism be able to operate locally.

2 An Alternative Technique in PtolemyII

PtolemyII [DIGH+98] is a heterogeneous concurrent modelling system; its objective is to provide a general framework for the construction and interoperability of executable models that are built under a wide variety of models of computation. In PtolemyII, models are constructed as a set of interacting components that can be easily designed to interact in a number of ways. A model of computation defines the semantics for this interaction.

PtolemyII provides a very general structural notation to build relationships between components of a model: topology, entity, relation, receiver, port, connection and link. This structural notation enables the construction of hierarchically composed domains. A domain is a package that implements a given model of computation.

Actors are components with input and output, that at least conceptually operate concurrently with other actors. Actors are defined by a set of action methods, that specify the action performed by the actor, and ports, that define the communication interface with the actor. The first action to be invoked is `initialize()`, which is invoked exactly once for the purpose of initializing state variables in the actor. Initialization is followed by any number of iterations. An iteration will typically involve a pre-fire to determine if firing is required or possible, one or many firings, and a post-fire to update persistent state and determine whether execution of the actor is complete. Only domains with fixed-point semantics (where multiple firings are required for the result of an iteration to converge) require multiple firings. A `wrapup()` method is invoked exactly once, typically for displaying final results.

Composite actors are collections of actors, and a director that governs the execution of its actors. The director is responsible for the local execution of the composite by observing a firing sequence for the actors of which the entity is composed. A manager is defined to govern the overall execution of the model through a top-level composite actor.

PtolemyII includes a process network domain implemented by Goel[Goe98]. During initialization, the process network domain creates a `ProcessThread` for each actor, then starts these threads on the first pre-fire. Unlike most other domains, the `ProcessThread` is responsible for the pre-fire, fire and post-fire of the actor. These methods are executed autonomously by this thread, with the director acting as an observer of the state of the threads and their actors. The `fire` method of the process network director performs no firing, but observes the network for cessation of progress due to a network stop or deadlock.

We are particularly interested in the PtolemyII mechanics for mutation. PtolemyII has a support kernel for operations on the topology itself. This is entirely separate from the expression of how the model runs (i.e. the kernel elegantly separates concerns of structure (syntax) from operation (semantics)).

PtolemyII utilizes its own mechanism for ensuring safe concurrent access to a running topology; each entity in the topology is immutably associated with a workspace, and appropriate read or write locks must be obtained on the workspace to mutate the network structure. The drawback is that the topology changes necessary for mutation must take place through kernel management methods. An actor that is mutating needs a mechanism to signal back to the kernel that it wants to mutate, and specify what is involved in the mutation. In domains where actors essentially execute one operation for each firing, there are well defined points, between actor firings, where mutation can take place.

This is not so for process networks; actors adhere to the firing rules, but are essentially autonomous so methods must be provided to allow "breaking out" of the their firing sequence. PtolemyII provides a mechanism through the manager for propagating a "stop firing" signal to all directors. In addition, the process network director issues a `stopThread()` to all threads to stop their firing. When the process network director observes all actors are either stopped, read-blocked, or in the process of mutation, control is returned back to manager. If the manager is required to iterate again, it will coordinate the execution of all mutations requested by actors with the workspace to ensure one mutation is only ever active, pre-fire the network by invoking `start()` on newly formed threads resulting from the mutation, and `restartThread()` on existing threads so they may continue.

The Sieve of Eratosthenes implemented by Goel is different to that implemented by [KM77]. This implementation uses a combined sieve-filter actor[1]. This actor has the job of filtering out a given prime. The first token received that is not a multiple of the prime it is filtering is a discovered prime. In reaction to this, the actor will request the director to perform a mutation to insert a new sieve-filter after itself. Actors request mutations using a `ChangeRequest` object with an `execute()` method that directly manipulates the topology. The code for this mutation is similar in purpose to that shown in Fig. 2. The director queues the request with its manager, and it is the queuing of this request that causes the manager to issue a stop fire signal. Unlike most other directors, the process network director blocks the actor requesting the change. This blockage leaves the actor with the impression of immediate execution of the change request.

3 Summary

We have considered process network reconfiguration, and the mutations of representation required for its implementation. We have looked at aspects of demand-driven (conservative) and data-driven (eager) evaluation strategies; centralized and localized representation management; and amenability to distribution of process network processes and channels.

Demand-driven evaluation lets us precisely identify points of activity in a process network. In [Wen82] we showed how mutations of centralized quasi-parallel and localized distributed representations could then be safely managed. When we allow more eager evaluation strategies, there are many more points of activity, and it becomes more difficult to specify conditions for safe mutation.

PtolemyII implements process networks as one domain among many; its primary objective is to facilitate system modelling combining many different domains. Hence, it adopts a centralized approach, which discourages distributed implementation. Further, it assumes eager evaluation[2]. This makes it difficult to precisely characterize and reason about the state of process network processes and channels involved in a reconfiguration, and is susceptible to problems with runaway process network fragments.

Our current work is examining a potentially distributed implementation utilizing conservative, eager and hybrid evaluation strategies with localized representation management. We believe this combination is essential for practical uses of process networks. In this paper, we have outlined a safe implementation scheme for reconfiguration in that situation.

References

[DIGH+98] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muiadi, S. Neuendorffer, J. Reekie, N. Smyth, and Y. Xiong. Heterogeneous Concurrent Modeling and Design in Java, PtolemyII Design Document. Technical report, EECS, University of California at Berkeley, 1998. http://ptolemy.eecs.berkeley.edu/publications/papers/98.

[Goe98] M. Goel. Process Networks in Ptolemy II. MS Report ERL Technical Report UCB/ERL No. M98/69, University of California, Berkeley, CA 94720, December 1998.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress 74*, pages 471–475. North Holland Publishing Company, 1974.

[KM77] G. Kahn and D.B. MacQueen. Coroutines and Networks of Parallel Processes. In B. Gilchrist, editor, *Proceedings of IFIP Congress 77*, pages 993–998. North Holland Publishing Company, 1977.

[VWW] J. Vayssière, D.L. Webb, and A.L. Wendelborn. Distributing Process Networks. Technical report, University of Adelaide. http://www.cs.adelaide.edu.au/ dpn/documents.html, to appear.

[VWW99] J. Vayssière, D.L. Webb, and A.L. Wendelborn. An Object-Oriented API for Process Networks. Technical Report 9904, University of Adelaide, October 1999. http://www.cs.adelaide.edu.au/ dpn/documents.html.

[Wen82] A.L. Wendelborn. Reconfiguration in the Process Networks of Kahn and MacQueen. *Australian Computer Science Communications*, 4(1):233–243, February 1982.

[1] it is not clear if it is possible to insert a new actor in front of an existing actor

[2] we believe is would be possible to implement conservative evaluation as a PtolemyII domain, and we are investigating how that may be done.