In Conference on Formal Methods and Models for Codesign (MEMOCODE)
June 22-25, 2004

Hierarchical Reconfiguration of Dataflow Models

Stephen Neuendorffer and Edward Lee
EECS Department
University of California at Berkeley
Berkeley, CA 94720, U.S.A.

Abstract

This paper presents a unified approach to analyzing patterns of reconfiguration in dataflow graphs. The approach is based on hierarchical decomposition of the structure and execution of a dataflow model. In general, reconfiguration of any part of the system might occur at any point during the execution of a model. However, arbitrary reconfiguration must often be restricted, given the constraints of particular dataflow models of computation or modeling constructs. For instance, the reconfiguration of parameters that influence dataflow scheduling or soundness of data type checking must be more heavily restricted. The paper first presents an abstract mathematical model that is sufficient to represent the reconfiguration of many types of dataflow graphs. Using this model, a behavioral type theory is developed that bounds the points in the execution of a model when individual parameters can be reconfigured. This theory can be used to efficiently check semantic constraints on reconfiguration, enabling the safe use of parameter reconfiguration at all levels of hierarchy.

1 Introduction

Dataflow models of computation [8, 10, 16] have been used to represent a wide variety of computing systems, such as signal processing algorithms [18], distributed computing workflows [21, 17], and embedded processing architectures [12, 20]. In a dataflow model, a computation is decomposed into components (called *actors*) that communicate by sending data values (called *tokens*) through *ports*. Ports are connected to other ports by communication *channels* that mediate the passage of tokens. Actors are not allowed to share state, so the only way for them to communicate is by exchanging tokens. Typically a channel is a queue that connects a single sending actor to a single receiving actor. These models are appealing since they closely match a designer's conceptualization of a system as a block diagram. The behavior of an actor can also be specified using another dataflow model, allowing high-level models to be refined into arbitrarily detailed ones. A hierarchically refined actor will be referred to as a *composite actor* when necessary.

Dataflow models of computation are also appealing because they offer opportunities for efficient implementation. Because actors only communicate through ports and do not share state, the parallelism of a system is completely exposed in the model and concurrent execution can be more easily understood. Additionally, under certain constraints, many dataflow models can be statically scheduled to run in bounded memory, which is critical for embedded system implementation. For instance, the synchronous dataflow model of computation [15, 4] breaks the execution of actors into a (possibly infinite) number of *firings* and requires that the number of tokens produced and consumed on each channel by an actor is fixed and known at scheduling time. Under these conditions, a finite schedule of actor firings can often be found such that the schedule can be executed an infinite number of times while using only a finite amount of memory for communication.

The communication interface consisting of an actor's ports also allows actors to be developed independently and provided as reusable library elements. Reusable library actors are commonly associated with *actor parameters* that alter the behavior of the actor. For instance, an actor representing a finite-impulse response (FIR) filter might have a parameter that determines the filter taps. The same actor might also provide multi-rate capabilities for efficient upsampling and downsampling, with corresponding parameters to determine the number of tokens produced and consumed during each execution of the filter. At design time, parameters help keep the size of actor libraries manageable and allow models to be quickly modified or tuned for performance. At run time, actor parameters allow for dynamic

This research is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), the Defense Advanced Research Projects Agency (DARPA), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from the State of California MICRO program, and the following companies: Daimler-Chrysler, Hitachi, Honeywell, Toyota and Wind River Systems.

reconfiguration of actors (and models) while a model is running.

There are many signal processing applications that can make use of dynamically reconfigured dataflow models. For instance, a communication system with adaptive echo cancellation can be modeled using dynamic reconfiguration of a parameterized filter. At a coarser level of granularity, the communication system might operate in two modes, a training mode and a communication mode. In the training mode, the system communicates a predetermined bit sequence and estimates the characteristics of the channel. These characteristics are used in the communication mode to improve the bit-error performance of the modem. The transition of training mode to communication mode can be modeled as system reconfiguration.

However, it is important to notice that in a synchronous dataflow model not all actor parameters can be reconfigured at run time. In particular, parameters that determine the number of tokens produced and consumed cannot be arbitrarily reconfigured without invalidating the static schedule. In the case of the multi-rate FIR filter mentioned previously, the parameter that determines the filter taps can be changed during execution without changing the number of tokens produced and consumed by the filter. On the other hand, the parameters that determine the upsampling and downsampling factors cannot be changed without concern for the validity of the schedule. This distinction represents a significant challenge to the uniform representation of reconfiguration in design tools.

This paper presents several modeling syntaxes for representing reconfiguration of parameters and a generic, hierarchical model of parameters and reconfiguration consistent with those syntaxes. The model allows parameter values to change at constrained points in the execution of the hierarchical model. These points, called *quiescent points*, are associated with actors in the hierarchical model, called *change contexts*, and are structured according to the hierarchical execution of the dataflow model. The *least change context* of a parameter determines a bound on how frequently the parameter is reconfigured. This bound can be used to simply express and efficiently check constraints on the reconfiguration of individual parameters. In particular, we describe how the least change context can be used to check constraints for generating parameterized synchronous dataflow schedules in a purely hierarchical framework [2].

2 Reconfiguration and Dataflow Scheduling

A synchronous dataflow model is a dataflow model where the token rate of each port is known *a priori*. In many cases, the token rate of a port depends on the value of actor parameters that do not change during the execution of the model. The token rates are used to pre-compute a finite sequence of actor firings that can be executed forever in bounded memory. This sequence is called a *static execution schedule*. By compiling this execution schedule into runtime code, efficient embedded software can be synthesized from synchronous dataflow graphs. A graphical representation of a hierarchical synchronous dataflow model is shown in Figure 1.

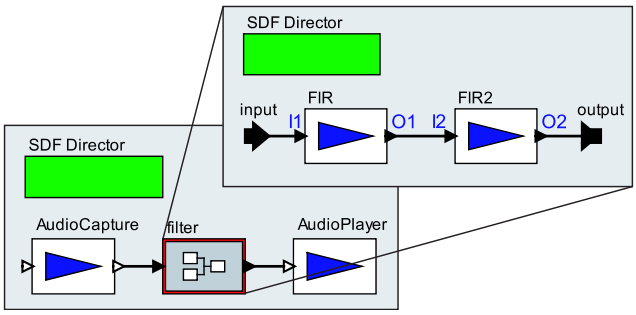

Figure 1. An example of a simple hierarchical synchronous dataflow model in Ptolemy II. The filter component is hierarchically decomposed into two multi-rate FIR filters with input and output token rates shown on the figure. The synchronous dataflow scheduler uses these rates to compute the number of tokens consumed and produced by the filter actor.

Strictly speaking, reconfiguration in a synchronous dataflow model must not affect the token rate of each port. Without this restriction reconfiguration might violate the properties of the static schedule, possibly causing deadlock or memory overflow. For instance, the model in Figure 1 depicts a pair of sample rate conversion filters that interface to an audio input device and an audio output device. The FIR filter actors in this model must be reconfigured to match hardware devices with different rates. This reconfiguration includes not only the filter taps, but also the decimation and interpolation factors of the filters that determine the number of tokens each actor produces and consumes. To allow reconfiguration of token rates, many extensions to the basic synchronous dataflow model have been proposed.

One approach to allowing reconfiguration of token rates involves the use of less constrained dataflow models that allow for rate parameters to change. For instance, the boolean-controlled dataflow model [6, 5] allows the rates of actors to change in response to external control inputs. Different combinations of control inputs effectively represent different 'states' of reconfiguration. However, from a designer's perspective interpreting combinations of control inputs as implicit configuration state is rather difficult. Additionally, the relaxation of dataflow constraints makes

static scheduling undecidable, although algorithms exist for computing static schedules in most practical cases.

Another approach to reconfiguration is to explicitly represent each configuration state as the state of an extended finite state machine or *modal model*, as in the *-charts model [9, 14], the FunState model [20], and the Stream-Based Functions model [13]. Each state of the finite state machine contains a dataflow model that is *active* in that particular state. Essentially, the active dataflow model replaces the finite state machine until the state machine makes a state transition. This approach is also practically limited by the number of configuration states that a designer can specify explicitly. Although static scheduling for these models is generally undecidable, scheduling properties such as deadlock freedom can be preserved. For instance, the heterochronous dataflow model only allows reconfiguration between executions of the toplevel schedule and can guarantee deadlock freedom [9]. If the number of configurations states is known beforehand then every possible schedule can be statically computed, although in practice the number of static schedules is often very large. In such cases, it is sometimes preferable to compute execution schedules "on-the-fly" as token rates change even if static schedules could theoretically be computed.

A third approach to allowing reconfiguration of token rates is to provide syntactic mechanisms for run-time modification of parameter values, as in the parameterized dataflow model [2]. The parameterized dataflow model distinguishes certain portions of a dataflow model as "initialization" graphs, which are capable of modifying the parameter values of the main part of the dataflow graph. In many cases, static scheduling [1] can still be performed by representing token rates symbolically and generating a symbolic or *quasi-static* schedule. A quasi-static schedule contains conditional or iterative constructs that cannot be determined statically at design time. Although the execution of this schedule depends on token rates that might change at run-time, the schedule is statically determined and can be compiled into efficient executable software. The key scheduling constraint is that actors in a parameterized synchronous dataflow model must be *locally synchronous* [3]. The local synchrony condition requires that although actor token rates may change, they are constant over the execution of a parameterized schedule. Fundamentally, the parameterized dataflow approach extends the heterochronous dataflow model to allow limited reconfiguration at all levels of hierarchy and feasible static scheduling.

3 Hierarchical Reconfiguration

In this paper, we will consider the semantic constraints on reconfiguration, without focusing on the largely syntactic differences in the above approaches. Our semantic basis for describing reconfiguration is the notion of *quiescent points* in the execution of a model, which occur after the firing of any actor. This section presents several modeling syntaxes for generically representing reconfiguration in hierarchical dataflow models. Each of these hierarchical syntaxes is essentially equivalent with respect to quiescent points and is independent of particular dataflow model of computation. In order to guarantee that each reconfiguration does not occur at quiescent points that violate dataflow constraints, such as local synchrony, we will rely on analysis of reconfiguration as described in Section 6.

The first syntax we present is based on an extended version of a modal model where the action associated with a finite state machine transition can set the value of actor parameters. During each firing of a modal model, the dataflow model associated with the active state is fired once and it communicates directly with the external ports of the modal model. After the active dataflow model is fired, the guard of each transition originating in the active state is evaluated. If exactly one guard is true, then that transition is taken and the destination state of the transition will be active in the next firing. If no guard is true, then the active state will remain active in the next firing. If multiple guards are true, then either the model is considered incorrect or one of the transitions can be chosen non-deterministically. If a transition is taken then the action of the transition is performed, possibly resulting in reconfiguration of a model parameter at the quiescent point after the firing. An example model is shown in Figure 2 and a plot from running the model in Figure 3.

The second syntax ties reconfiguration to dataflow in a model. Reconfiguration in this model is represented by *reconfiguration ports*, a special form of dataflow input port. An example of this syntax is shown in Figure 4. Each reconfiguration port is bound to a parameter of its actor and tokens received through the port reconfigure the parameter. More specifically, a firing of an actor with reconfiguration ports is composed of two distinct sub-firings separated by an internal quiescent state. During the first sub-firing the actor consumes a single input token only from reconfiguration ports. The input tokens determine the reconfiguration applied during the internal quiescent state. During the second sub-firing input tokens are consumed from normal dataflow input ports, computation is performed, and any outputs are produced. For a composite actor, contained actors are not fired during the first sub-firing and the associated dataflow model is executed only during the second sub-firing. Reconfiguration ports exist in many dynamically-scheduled dataflow environments, such as AVS/Express (Advanced Visual Systems, Inc.).

A third syntax represents reconfiguration using a special actor, the `setParameter` actor. This actor has a single input port and is bound to a parameter of the containing

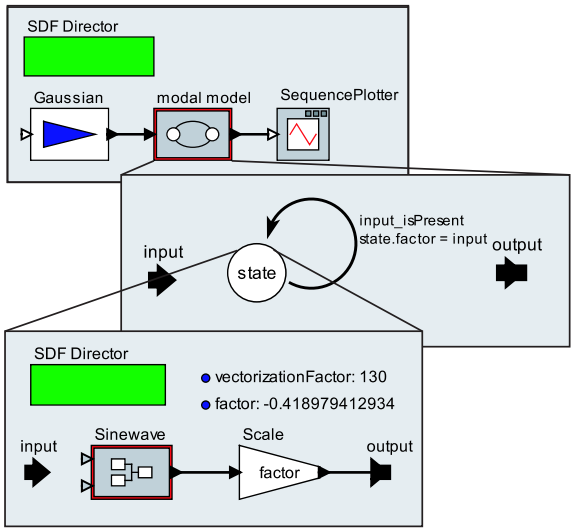

Figure 2. A graphical representation of a simple modal model in Ptolemy II showing three levels of hierarchy. In this model, the contained model is executed first producing a block of output tokens. After producing output tokens, the modal model transition is performed, since the guard is always true, resulting in reconfiguration of the contained model for the next block. In this model, reconfiguration results in sinewave segments with different amplitudes. The parameters of the interior dataflow model ensure that 130 samples of the sinewave are generated in each block.

model. The actor consumes a single token during each firing of the `setParameter` actor and the bound parameter is reconfigured during the quiescent point after the firing. Although the `setParameter` actor might appear similar to a reconfiguration port, it allows for a parameter to be more frequently reconfigured, since the `setParameter` actor might fire more than once in the execution schedule of its contained model. The result is that it is often easier for a designer to violate dataflow scheduling constraints using the `setParameter` actor than with the other two syntaxes.

It is important to notice that each of these syntaxes is constrained in the set of parameters that can possibly be reconfigured. In the case of the modal model, only parameters referenced in state transitions can be reconfigured by the modal model. Reconfiguration ports and the `setParameter` actor are bound to a single parameter, and the binding cannot change at run-time. This restriction is crucial to the useful application of the reconfiguration analysis proposed in Section 6.

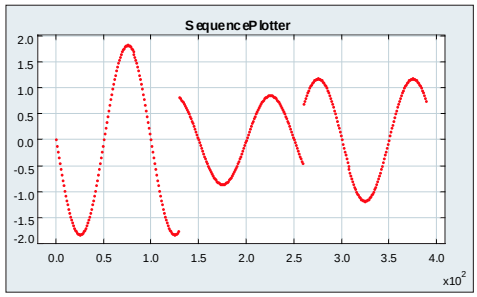

Figure 3. A plot from running the model shown in Figure 2.

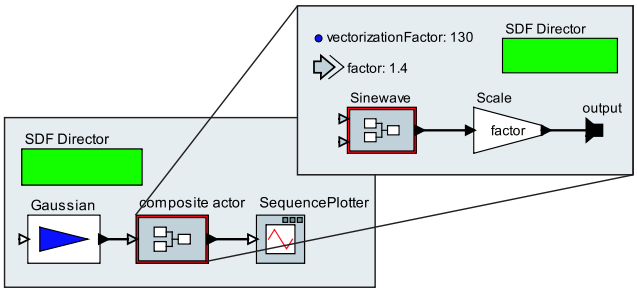

Figure 4. A graphical representation of a simple model with a reconfiguration port in Ptolemy II. In this model, the reconfiguration port is shaded grey instead of black, and reconfigures the parameter named "factor" directly to its right. This model behaves essentially identically to the one in figure 2, except that the reconfiguration occurs prior to each block of samples being produced rather than after.

4 Parameterization Model

In this section, we present an abstract mathematical model for hierarchical dataflow models of computation with parameterization and reconfiguration. This model allows reconfiguration at all levels of the hierarchy, but does not bind reconfiguration to specific syntactic constructs. The model uniformly represents static schedules, quasi-static schedules, token rates, and user-level configuration options as *actor parameters*. The model explicitly represents the dependencies between parameters. These dependencies may arise from a variety of sources, such as an expression in a design environment that expresses the value of parameter in terms of another, a declaration of token rates in a library actor, or a scheduler that synthesizes a schedule and corresponding token rates for the external ports of a model.

A *hierarchical reconfiguration model* is represented by a finite tree of actors, called the *containment tree*. Leaf elements of the tree are primitive, or *atomic* actors, and non-leaf elements are called *composite* actors. The root of the containment tree is the *toplevel* composite actor. The behavior of a composite actor is given by a dataflow model consisting of the actors that are its direct children in the tree. The dataflow model associated with each composite actor is assumed to reference *external ports* that communicate with the dataflow model that contains the composite actor. The composite actor at the root of the containment tree contains no external ports. We say that the all actors in a subtree are *contained* by the root of the subtree. Similarly, a composite actor *contains* all actors in the subtree rooted by the composite actor, including itself.

Formally, the set of actors in a model is $\mathbf{A}$. We write $c \trianglerighteq a$ if the composite actor c contains a. The relation $\trianglerighteq \subseteq \mathbf{A} \times \mathbf{A}$ is a transitive, reflexive, antisymmetric partial order and $(\mathbf{A}, \trianglerighteq)$ is a tree. A fundamental property of the containment tree is that the set of actors that contain a particular actor in the tree form a *chain*. Or, more formally, $\forall c \in \mathbf{A}$ the set $\{x \in \mathbf{A} | x \trianglerighteq c\}$ is totally ordered by $\trianglerighteq$.

Every actor a has a set of parameters P_a that determines the dataflow behavior of actor a. Different actors are associated with disjoint parameter sets ($a \neq b \implies P_a \cap P_b = \emptyset$), allowing them to be independently configured. The unique actor associated with a parameter p is $actor(p)$. The finite set of all parameters in the model is $\mathbf{P} = \bigcup_{a \in \mathbf{A}} P_a$. The value of each parameter at any point during execution of a model is given by an element of the set $\mathbf{T}$ of tokens. A *valuation function* is a function in $\mathbf{P} \rightarrow \mathbf{T}$ that gives the value of each parameter in a model.

In practical models, the values of parameters are often dependent on one another. This dependence might be specified explicitly in the construction of a model, e.g., one parameter is given as an expression of another, or implicitly, e.g., a dataflow scheduler synthesizes some parameter values. We generally ignore these differences and take a denotational approach to describing constraints that parameter values must satisfy. We write that the value of a parameter p depends on a finite, indexed set of parameters $domain^p = \{domain^p_1, \ldots, domain^p_n\}$. We say that a parameter p is *independent* if $domain^p$ is empty, and *dependent* otherwise. Independent parameters in a model are allowed to be modified during reconfiguration.

The value of each dependent parameter p is constrained by a *constraint function* $constraint_p : \mathbf{T}^n \rightarrow \mathbf{T}$, where n is the number of elements in $domain^p$. A *consistent valuation function* is a valuation function where the value of every dependent parameter satisfies the parameter's constraint function.

Definition 1. *Consistent valuation function:*
A valuation function v is consistent if and only if $\forall p \in \mathbf{P}, p$ *is dependent* $\implies constraint_p(v(domain^p_1), \ldots, v(domain^p_n)) = v(p)$

For mathematical convenience, we define $\rightsquigarrow \subseteq \mathbf{P} \times \mathbf{P}$ to be the *dependence relation* between parameters. The dependence relation is the least transitive relation between parameters, such that $\forall x \in domain^p, x \rightsquigarrow p$. In order for a model to be well-defined, we require that the dependence relation is not reflexive (i.e., $\forall x \in \mathbf{P}, x \not\rightsquigarrow x$). The set of parameters that are transitively modified by a parameter p will be written $\overset{\rightsquigarrow}{p} = \{x \in \mathbf{P}, p \rightsquigarrow x\}$, and for a set of parameters P, $\overset{\rightsquigarrow}{P} = \bigcup_{p \in P} \overset{\rightsquigarrow}{p}$. Generally speaking, a design tool will determine the values of parameters in the set $\overset{\rightsquigarrow}{p}$ based on the value of a parameter p.

5 Reconfiguration Semantics

This section describes an abstract semantics for hierarchical reconfiguration models. The semantics is defined incompletely in order to encompass various dataflow models, scheduling techniques, and heterogeneous compositions of different models. Primarily, the dataflow model associated with each composite actor in the model is assumed to be *reactive* and *hierarchically composed*. Reactivity requires that the behavior of each actor consists of a totally ordered sequence of *firings*. During the firing of an actor, it may send and receive data from communication channels and perform computation. Between firings, an actor is *quiescent* and cannot communicate or perform computation. Hierarchical composition requires that each actor firing is encompassed by a single firing of its container. Equivalently, hierarchical composition requires that when a composite actor is quiescent, all actors deeply contained by the composite actor are also quiescent.

Formally, we write the set of all quiescent points of actor a during an execution of a model as Q^a where $c \trianglerighteq a \implies Q^c \subseteq Q^a$. The set $\mathbf{Q} = \bigcup_{a \in \mathbf{A}} Q^a$ is the set of all quiescent points of all actors. The *precedence relation* is a transitive, reflexive, antisymmetric partial order $\leq \subseteq \mathbf{Q} \times \mathbf{Q}$ that gives a time-ordering of quiescent points. The precedence relation is constrained such that the quiescent points Q^a of an actor a are totally ordered by $\leq$. If $q_1 \leq q_2$ then the quiescent point q_1 always occurs before q_2. If $q_1 \not\leq q_2$ and $q_2 \not\leq q_1$ then there is freedom in the execution of q_1 and q_2, possibly allowing for concurrent execution. An illustration of quiescent points is shown in Figure 5.

In addition to constraining the dataflow behavior of a model, quiescent points in the execution also form points where reconfiguration is allowed to occur. At each quiescent point q in the execution of a model, a set of indepen-

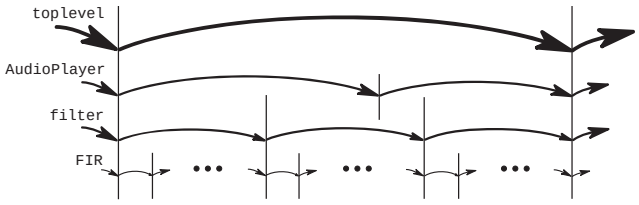

Figure 5. A graphical representation of the quiescent points in one execution of the model in Figure 1. The model is one where actor `toplevel` contains actors `AudioPlayer` and `filter`, and actor `filter` in turn contains actor `FIR`. Quiescent points are shown as vertical lines and actor firings are shown as arrows. A quiescent point is a quiescent point of an actor if a firing arrow of the actor starts or end at the quiescent point. The direction of arrows represents the partial ordering of quiescent points.

dent parameters $R(q)$ is selected for reconfiguration. Based on this initial set of parameters and reconfigured values, reconfigured values for dependent parameters in $\overset{\rightsquigarrow}{R(q)}$ are determined based on their individual constraints and those parameters are also reconfigured.

Note that the set of parameters $\overset{\rightsquigarrow}{R(q)}$ reconfigured at a particular quiescent point may be associated with actors anywhere in the model. However, dataflow semantics may require that certain parameters are not reconfigured, or are reconfigured only at certain quiescent points in the execution of the model. For example, static scheduling often requires limited reconfiguration. A bound on the quiescent points when a parameter is reconfigured is established by two notions of a constant parameter. The following two theorems give intuition about constant parameters.

Definition 2. *Constant parameter:*
Parameter p is constant *if and only if*
$\forall a \in \mathbf{A}, \forall q \in Q^a, p \notin \overset{\rightsquigarrow}{R(q)}$.

Definition 3. *Constant parameter over actor firings:*
Parameter p is constant over firings of actor c *if and only if*
$\forall a \in \mathbf{A}, \forall q \in Q^a, p \in \overset{\rightsquigarrow}{R(q)} \implies q \in Q^c$.

Theorem 1. *p is constant implies p is constant over firings of any actor.*

Proof. Let c be an arbitrary element of $\mathbf{A}$
$\forall x \in \mathbf{A}, \forall q \in Q^x, p \notin \overset{\rightsquigarrow}{R(q)}$
$\forall x \in \mathbf{A}, \forall q \in Q^x, p \in \overset{\rightsquigarrow}{R(q)} \implies q \in Q^c$
p is constant over firings of c
□

Theorem 2. *p is constant over firings of c and $c \trianglerighteq a$ implies p is constant over firings of a .*

Proof. $\forall x \in \mathbf{A}, \forall q \in Q^x, p \in \overset{\rightsquigarrow}{R(q)} \implies q \in Q^c$
$Q^c \subseteq Q^a$
$\forall x \in \mathbf{A}, \forall q \in Q^x, p \in \overset{\rightsquigarrow}{R(q)} \implies q \in Q^a$
p is constant over firings of a
□

Constant parameters are not reconfigured during a particular execution of the dataflow model. Type parameters that are used for static data type checking are commonly required to be constant parameters in order to guarantee type soundness. Special parameters that determine the structure of the model, such as the number of replications of a single actor, and parameters that are partially evaluated by a code generation system are also required to be constant. The reconfiguration constraint of heterochronous dataflow models requires that any parameter representing the dataflow schedule of a composite actor is constant over firings of the toplevel composite actor. The local synchrony constraint for parameterized synchronous dataflow scheduling requires that the parameter representing the execution schedule of a composite actor c is constant over firings of c.

6 Change Contexts

In general, it is undecidable to determine if a parameter is constant or constant over firings of an actor on any particular execution since the set $\mathbf{Q}$ is infinite and $R(q)$ for $q \in \mathbf{Q}$ might depend on data given to a model only at runtime. However, approximating over all the possible behaviors results in a simple and intuitive approximation can be decidably checked. When combined with suitably formulated safety requirements on reconfiguration, these approximations form a *behavioral type theory for reconfiguration*.

In order to statically analyze the reconfiguration of a model, we will concentrate on approximately analyzing all possible reconfigurations of a model during any execution of a model. To begin with, we assume that the reconfiguration model includes a set $R^a \subseteq \mathbf{P}$ for every actor a. The set R^a is the smallest set that contains all independent parameters that may be modified when actor a is quiescent. During all executions of the model, $\forall a \in \mathbf{A}, \forall q \in Q^a, R(q) \subseteq R^a$, and $\overset{\rightsquigarrow}{R(q)} \subseteq \overset{\rightsquigarrow}{R^a}$. For convenience, we say that an actor a is a *change context* for all parameters in R^a, and that a parameter is inherently constant (or inherently constant over actor firings) if its change contexts satisfy certain constraints. Intuitively, a parameter is inherently constant (over actor firings) if it is constant (over actor firings) during all executions of the model.

Definition 4. *Change context:*
An actor a is a change context of a parameter p, written $a \rightsquigarrow p$, if and only if $p \in \overset{\rightsquigarrow}{R}^a$.

Definition 5. *Inherently constant parameter:*
Parameter p is inherently constant *if and only if* $\forall a \in \mathbf{A}, a \not\rightsquigarrow p$.

Definition 6. *Inherently constant parameter over actor firings:*
Parameter p is inherently constant over firings of actor a *if and only if* $\forall c \in \mathbf{A}, c \rightsquigarrow p \implies c \trianglerighteq a$.

Theorem 3. *p is inherently constant implies p is constant.*

Proof. $\forall z \in \mathbf{A}, z \not\rightsquigarrow p$
$\forall z \in \mathbf{A}, p \notin \overset{\rightsquigarrow}{R}^z$
$\forall z \in \mathbf{A}, \forall q \in Q^z, \overset{\rightsquigarrow}{R}(q) \subseteq \overset{\rightsquigarrow}{R}^z$
$\forall z \in \mathbf{A}, \forall q \in Q^z, p \notin \overset{\rightsquigarrow}{R}(q)$
p is constant

□

Theorem 4. *p is inherently constant over firings of actor c implies p is constant over firings of actor c.*

Proof. $\forall x \in \mathbf{A}, x \rightsquigarrow p \implies x \trianglerighteq c$
$\forall x \in \mathbf{A}, p \in \overset{\rightsquigarrow}{R}^x \implies x \trianglerighteq c$
$\forall x \in \mathbf{A}, \forall q \in Q^x, \overset{\rightsquigarrow}{R}(q) \subseteq \overset{\rightsquigarrow}{R}^x$
$\forall x \in \mathbf{A}, \forall q \in Q^x, p \in \overset{\rightsquigarrow}{R}(q) \implies x \trianglerighteq c$
$x \trianglerighteq c \implies Q^x \subseteq Q^c$
$\forall x \in \mathbf{A}, \forall q \in Q^x, p \in \overset{\rightsquigarrow}{R}(q) \implies q \in Q^c$
p is constant over firings of actor c

□

According to the previous definitions, it is decidable to automatically check whether a parameter is inherently constant or inherently constant over the firings of an actor. For instance, a direct implementation of the above definitions might compute the set $\overset{\rightsquigarrow}{R}^a$ for each actor and check the constraint for each parameter. Unfortunately, in large hierarchical models the toplevel parameters often have many dependent parameters deep in the hierarchy and a direct implementation performs significant redundant computation by computing $\overset{\rightsquigarrow}{R}^a$ independently. A more efficient algorithm could compute $\overset{\rightsquigarrow}{R}^a$ simultaneously for each actor, iteratively updating each set. However, in large models, the memory usage of this technique becomes large, since all of the sets must be stored in memory at the same time.

The rest of this section presents an approximate and efficient algorithm that can verify whether parameters in a model are constant. The intuition behind this algorithm is that the set $\{a \in \mathbf{A} : a \rightsquigarrow p\}$ of all change contexts of a parameter p can often be approximated by the *greatest lower bound* of the set. The greatest lower bound, written $\sqcap A$, of a set is the unique element that is a lower bound for the set (i.e., is less than every element in the set), and also greater than every other lower bound [7]. We note that the greatest lower bound of a subset of $\mathbf{A}$ does not necessarily exist. In order to guarantee that the approximation always exists, the algorithm computes the greatest lower bound in an artificially constructed ordered set $\mathbf{A}_\perp^\top$. This set contains a special element $\perp$ to represent the case when the greatest lower bound does not exist, and a special element $\top$ to represent the greatest lower bound of an empty set.

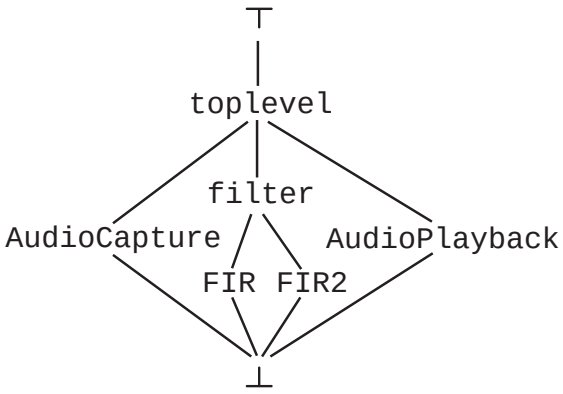

Figure 6. An example of the lattice formed by augmenting the containment tree of the model in Figure 1 with artificial top and bottom elements.

Formally, the set $\mathbf{A}_\perp^\top$ is defined to be $\mathbf{A} \cup \{\top, \perp\}$ where $\top$ and $\perp$ are artificial elements not in $\mathbf{A}$. The ordering relation $\trianglerighteq_\perp^\top \subseteq \mathbf{A}_\perp^\top \times \mathbf{A}_\perp^\top$ is defined to be the transitive, reflexive, antisymmetric ordering relation where $\forall a \in \mathbf{A}, \forall b \in \mathbf{A}, a \trianglerighteq b \iff a \trianglerighteq_\perp^\top b$ and $\forall a \in \mathbf{A}_\perp^\top, \top \trianglerighteq_\perp^\top a \trianglerighteq_\perp^\top \perp$. With this construction, $(\mathbf{A}_\perp^\top, \trianglerighteq_\perp^\top)$ is a *lattice* [7]. A basic property of a lattice is that every set of elements A in the lattice has a greatest lower bound in the lattice. An example of a resulting lattice is shown in Figure 6.

We define the function $\lfloor \cdot \rfloor : \mathbf{P} \to \mathbf{A}_\perp^\top$ as shown in Definition 7 and say that $\lfloor p \rfloor$ is the *least change context* of the parameter p. The least change context of a parameter p is essentially a conservative approximation of the set of all the change contexts of p. If the least change context of a parameter p is either $\top$ or an element of $\mathbf{A}$, then the set of change contexts of p is limited and reconfiguration of p can only occur during the quiescent points of certain actors. On the other hand, if the least change context is $\perp$ then the conservative approximation gives no interesting information about reconfiguration, and no restrictions on reconfiguration can be inferred. Theorems 5 and 6 prove the soundness of the least change context approximation.

Definition 7. *Least change context of a parameter:*
The least change context of a parameter p, $\lfloor p \rfloor$, is an element of $\mathbf{A}_\perp^\top$ where $\lfloor p \rfloor = \sqcap\{a \in \mathbf{A}_\perp^\top : a \in \mathbf{A} \wedge a \rightsquigarrow p\}$
Or equivalently,

$$\lfloor p \rfloor = \begin{cases} \top & \text{if } \{a \in \mathbf{A} : a \rightsquigarrow p\} = \emptyset \\ \sqcap\{a \in \mathbf{A} : a \rightsquigarrow p\} & \text{if } \{a \in \mathbf{A} : a \rightsquigarrow p\} \neq \emptyset \text{ and } \sqcap\{a \in \mathbf{A} : a \rightsquigarrow p\} \text{ exists} \\ \perp & \text{otherwise} \end{cases}$$

Theorem 5. $\lfloor p \rfloor = \top$ *implies p is inherently constant.*

Proof. $\{a \in \mathbf{A} : a \rightsquigarrow p\} = \emptyset$
$\forall a \in \mathbf{A}, a \not\rightsquigarrow p$
p is inherently constant
□

Theorem 6. $\lfloor p \rfloor \in \mathbf{A}$ *implies p is inherently constant over firings of $\lfloor p \rfloor$.*

Proof. $\lfloor p \rfloor = \sqcap\{a \in \mathbf{A} : a \rightsquigarrow p\}$
$\forall a \in \mathbf{A} : a \rightsquigarrow p \implies a \trianglerighteq \lfloor p \rfloor$
p is inherently constant over firings of $\lfloor p \rfloor$
□

Approximate approaches to static analysis must always balance usefulness with utility and avoid discarding interesting information about behavior. One source of approximation in our theory arises from the inherently constant property, which requires that reconfiguration of a parameter not occur during *any* behavior of the model. While it is possible to construct models that specify reconfiguration that does not actually occur, such as a modal model where the guards of transitions are always false, we accept that such models might be reflected by our approach. A second source of approximation arises from the least change context approximation to the set of change contexts. Theorem 7 shows that for interesting inherently constant constraints, such as the local synchrony constraint for parameterized synchronous dataflow scheduling, the least change context approximation does not discard information.

Based on the structure of a model, we notice that the least change context of a parameter must satisfy two simple constraints over the lattice $(\mathbf{A}_\perp^\top, \trianglerighteq_\perp^\top)$. The first constraint (Theorem 8) requires that the least change context of a parameter p cannot be any higher in the hierarchy than the least change context of a parameter that p depends on. The second constraint (Theorem 9) requires that if a parameter is reconfigured by an actor, then the actor must contain the least change context of the parameter. In fact, these constraints will be satisfied by not only the least change context but also *any* lower bound on the set of change contexts. Using the greatest lower bound, however, gives the most information about the set of change contexts for a parameter. By using these constraints, the least change context of a parameter can be computed *without direct computation of the set of change contexts for each parameter*. One algorithm for computing the solution is known to be linear time in the number of constraints [19]. The algorithm computes $\lfloor \cdot \rfloor$ by beginning with an initial guess where $\forall p \in \mathbf{P}, \lfloor p \rfloor = \top$. The initial guess is updated according to each constraint until all the constraints are satisfied.

Theorem 7. p *is inherently constant over* $actor(p)$ *implies that* $\lfloor p \rfloor \neq \perp$.

Proof. By cases.
Let p be an arbitrary element of $\mathbf{P}$
Case 1: $\nexists c \in \mathbf{A}$ such that $c \rightsquigarrow p$
$\implies \lfloor p \rfloor = \top$
Case 2: $\exists$ unique $c \in \mathbf{A}$ such that $c \rightsquigarrow p$
$\implies \lfloor p \rfloor = c$
Case 3: $\exists A \subseteq \mathbf{A}$ such that $\forall c \in A, c \rightsquigarrow p$
$\forall c \in A, c \trianglerighteq actor(p)$
$(A, \trianglerighteq)$ is a chain
$\exists c \in A, \forall x \in A, x \trianglerighteq c$
$\implies \lfloor p \rfloor = c$
□

Theorem 8. $p_1 \rightsquigarrow p_2$ *implies* $\lfloor p_1 \rfloor \trianglerighteq_\perp^\top \lfloor p_2 \rfloor$.

Proof. Let p_1 and p_2 be arbitrary elements of $\mathbf{P}$
$\forall a \in \mathbf{A}, a \rightsquigarrow p_1 \implies a \rightsquigarrow p_2$
$\{a \in \mathbf{A} : a \rightsquigarrow p_1\} \subseteq \{a \in \mathbf{A} : a \rightsquigarrow p_2\}$
$\{a \in \mathbf{A}_\perp^\top : a \in \mathbf{A} \wedge a \rightsquigarrow p_1\} \subseteq \{a \in \mathbf{A}_\perp^\top : a \in \mathbf{A} \wedge a \rightsquigarrow p_2\}$
$\sqcap\{a \in \mathbf{A}_\perp^\top : a \in \mathbf{A} \wedge a \rightsquigarrow p_1\} \trianglerighteq_\perp^\top \sqcap\{a \in \mathbf{A}_\perp^\top : a \in \mathbf{A} \wedge a \rightsquigarrow p_2\}$
$\lfloor p_1 \rfloor \trianglerighteq_\perp^\top \lfloor p_2 \rfloor$
□

Theorem 9. $p \in R^c$ *implies* $c \trianglerighteq_\perp^\top \lfloor p \rfloor$.

Proof. Let c be an arbitrary element of $\mathbf{A}$ and p be an arbitrary element of R^c
$p \in \overset{\leftrightsquigarrow}{R^c}$
$c \rightsquigarrow p$
$c \in \{a \in \mathbf{A} : a \rightsquigarrow p\}$
$c \in \{a \in \mathbf{A}_\perp^\top : a \rightsquigarrow p\}$
$c \trianglerighteq_\perp^\top \sqcap\{a \in \mathbf{A}_\perp^\top : a \rightsquigarrow p\}$
$c \trianglerighteq_\perp^\top \lfloor p \rfloor$
□

7 Design Example

Figure 7 shows an example signal processing model that illustrates some of the issues involved with reconfiguration. This model describes a *blind communication receiver* that must analyze and process a received signal with unknown characteristics. Specifically, the model analyzes an arbitrary pre-recorded segment of a digital Phase Shift Keyed

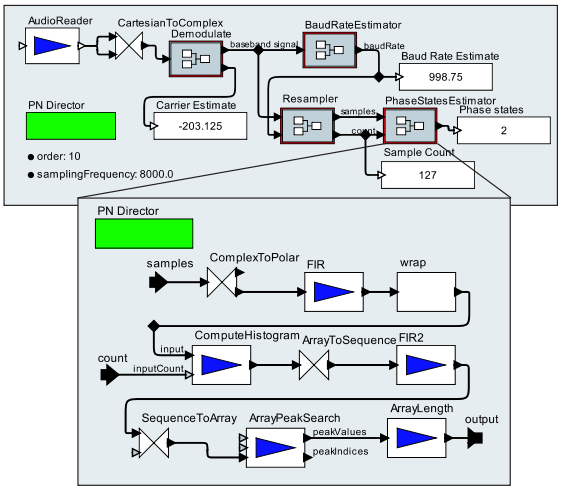

Figure 7. A process network design example where each actor is an independent thread that blocks waiting for input data.

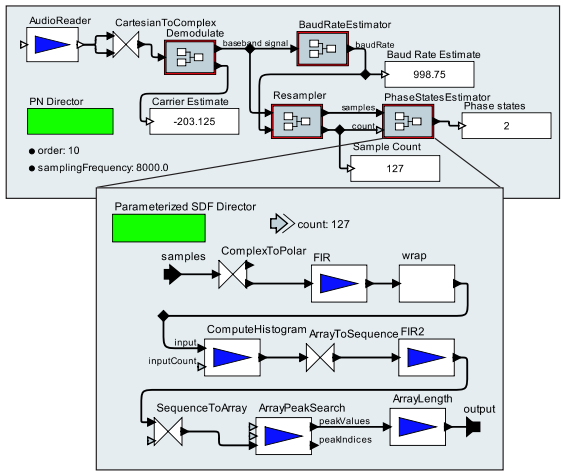

Figure 8. An improved design that allows more opportunities for static dataflow scheduling. The `count` port has been converted from a dataflow port to a reconfiguration port, and the `PhaseStatesEstimator` model has been changed to use a parameterized synchronous dataflow scheduler.

(PSK) signal to determine the carrier frequency, baud rate, and number of phase shifts of the signal. The model is a heterogeneous composition of synchronous dataflow models and parameterized synchronous dataflow models. The submodels are composed hierarchically in a *Kahn-MacQueen process network* [11, 16]. Each actor in the process network acquires an operating system thread and communicates with other other components through dynamically resized queues. Actor threads block until communication queues have enough data.

The `Demodulate` and `BaudRateEstimator` actors are implemented by synchronous dataflow models that process 2^{order} input samples and compute estimates of the carrier frequency and symbol rate of the input signal. Additionally, the `Demodulate` block synthesizes a carrier signal of the appropriate frequency and outputs a baseband version of the input signal. The `Resampler` actor samples the baseband signal at the estimated baudrate, and outputs a data-dependent number of complex samples. The `PhaseStatesEstimator` processes the resampled data to estimate the number of different phases used in the PSK transmission.

A hierarchical process network implementing the `PhaseStatesEstimator` is shown in detail in Figure 7. This model relies on the `ComputeHistogram` actor, which computes an array representing a histogram of input data. The number of samples used to compute the histogram is specified as an actor parameter bound to the `inputCount` reconfiguration port. The model is constructed so that the histogram computes a histogram of all the resampled data.

Overall, the data-dependent nature of the resampling operation prevents the entire model from being statically scheduled, since the number of resampled data tokens is not available to a scheduler. However, in order to avoid a large number of operating system threads, we would prefer a statically or quasi-static schedule for the `PhaseStatesEstimator`. An attempt to apply a synchronous dataflow scheduling model results in reconfiguration constraints that cannot be satisfied in the model, since the parameter that determines the number of tokens consumed by the histogram is reconfigured. Attempting to use a parameterized synchronous dataflow scheduling model also fails, since the least change context of a parameter representing the schedule would be the `ComputeHistogram`. This least change context implies that the schedule is constant over firings of `ComputeHistogram` but not constant over firings of `PhaseStatesEstimator`, as required. An attempt to construct such a model results in a reconfiguration type error.

One design solution is to modify the model as shown in Figure 8. In this model, reconfiguration has been moved up one level in the model, resulting in reconfiguration just before the `PhaseStatesEstimator` is fired. The value of the `inputCount` parameter is equal to value of the `count` parameter, which is reconfigured by a reconfiguration port. In this model, the schedule depends on the number of tokens consumed each firing by the input port

of the `ComputeHistogram` actor, which is determined by its `inputCount` parameter. As a result, the schedule is constant over firings of the `PhaseStatesEstimator`, satisfying the local synchrony constraint for parameterized synchronous dataflow schedules.

8 Conclusion

In this paper, we have presented a model of parameterization and reconfiguration for hierarchical dataflow models. The model assumes that reconfiguration of parameters occurs at quiescent points in the hierarchical execution of the model. The quiescent points at which a parameter is reconfigured are restricted by showing that the parameter is inherently constant over the firings of an actor in the model. Restrictions on reconfiguration are often necessary in order to ensure that semantic constraints of the model, such as the local synchrony constraint for parameterized synchronous dataflow scheduling, are satisfied.

In order to analyze the reconfiguration of a model and ensure semantic constraints are satisfied, we have presented a behavioral type theory that analyzes reconfiguration. The theory analyzes the change contexts in a model that perform reconfiguration and relies on two abstractions of the behavior of the model. Firstly, the theory analyzes the behavior of the model based on all possible executions of a model. If invalid reconfiguration might occur during any execution of the model, the theory assumes that the model is invalid. Secondly, the theory approximates the set of change contexts for a parameter by the least change context. The least change context approximation allows for efficient type checking, but might result in no information about reconfiguration. We show, however, that the least change context approximation is sufficient to check interesting semantic constraints.

References

[1] B. Bhattacharya and S. S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In *Proc. of the International Workshop on Rapid System Prototyping*. IEEE, June 2000.

[2] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, Oct. 2001.

[3] B. Bhattacharya and S. S. Bhattacharyya. Consistency analysis of reconfigurable dataflow specifications. In *Embedded Processor Design Challenges*, number 2268 in Lecture Notes in Computer Science, pages 1–17. Springer-Verlag, Oct. 2002.

[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.

[5] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California at Berkeley, CA, 1993.

[6] J. T. Buck and E. A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 429–432, Minneapolis, MN, 1993.

[7] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[8] J. B. Dennis. First version of a dataflow procedure language. In *Programming Symposium: Proceedings, Colloque sur la Programmation*, number 19 in Lecture Notes in Computer Science, pages 362–376. Springer, Apr. 1974.

[9] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999.

[10] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress 74*. North-Holland Publishing Co., 1974.

[11] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of the IFIP Congress 77*, pages 993–998, Paris, France, 1977. International Federation for Information Processing, North-Holland Publishing Company.

[12] A. Kalavade. *System-Level Codesign Of Mixed Hardware-Software Systems*. PhD thesis, EECS Department, University of California at Berkeley, CA, 1995.

[13] B. Kienhuis and E. F. Deprettere. Modeling stream-based applications using the SBF model of computation. *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 34(3):291–300, 2003.

[14] B. Lee. *Specification and Design of Reactive Systems*. PhD thesis, EECS Department, University of California at Berkeley, CA.

[15] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, pages 55–64, Sept. 1987.

[16] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–798, May 1995.

[17] B. Ludaescher, I. Altintas, and A. Gupta. Compiling abstract scientific workflows into web service workflows. In *Proc. of the Intl. Conference on Scientific and Statistical Database Management (SSDBM)*, 2003.

[18] J. Pino, S. Ha, E. Lee, and J. Buck. Software synthesis for DSP using Ptolemy. *Journal of VLSI Signal Processing*, Jan. 1995.

[19] J. Rehof and T. Mogenson. Tractable constraints in finite semilattices. In *Proc. of the Third International Static Analysis Symposium (SAS)*, number 1145 in Lecture Notes in Computer Science, pages 285–300. Springer-Verlag, Sept. 1996.

[20] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. FunState - an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, Aug. 2001.

[21] D. Webb, A. Wendelborn, and K. Maciunas. Process networks as a high-level notation for metacomputing. In *Proc. of the Int. Parallel Programming Symposium (IPPS), Workshop on Java for Distributed Computing*, 1999.