The Coroutine Model of Computation

Chris Shaver and Edward A. Lee

EECS Department, University of California Berkeley
{shaver,eal}@eecs.berkeley.edu

Abstract. This paper presents a general denotational formalism called the *Coroutine Model of Computation* for control-oriented computational models. This formalism characterizes atomic elements with control behavior as *Continuation Actors*, giving them a static semantics with a functional interface. *Coroutine Models* are then defined as networks of Continuation Actors, representing a set of control locations between which control traverses during execution. This paper gives both a strict and non-strict denotational semantics for Coroutine Models in terms of compositions of Continuation Actors and their interfaces. In the strict form, the traversal of control locations forms a control path producing output values, whereas in the non-strict form, execution traverses a tree of potential control locations producing partial information about output values. Furthermore, the given non-strict form of these semantics is claimed to have useful monotonicity properties.

1 Introduction

Let a *control-oriented model* describe a system characterized by a network of control locations traversed sequentially during execution. At each location visited during execution, an action may be performed that produces outputs or manipulates the state of the system. However, at each location there is also a determination of how the traversal through the network of locations will subsequently progress. This determination can depend on both the inputs to the system, as well as its state, and is often represented as a conditional or guard. In some control-oriented models this determination can also include the possibility of the model either *suspending* or *terminating* its thread of control in the context of a larger model or execution environment.

Examples of control-oriented models include traditional imperative programming models, control flow graphs, and automata-based models such as state machines or labeled transition systems. Languages such as Esterel [4], Reactive C [5], and StateCharts [10] are also control-oriented in this sense. In the case of Esterel or Reactive C, the control locations correspond to individual imperative statements in the language, with the traversal of these locations corresponding to the control flow of the language. In StateCharts control locations are simply states with traversal governed by the guards of transitions. These three examples have the additional feature of suspending and resuming control over a series of *reactions*, as defined by Boussinot [4]. In the **Ptolemy II** environment, Modal

Models [14] give a hierarchical layer of control-oriented behavior to heterogeneous models. Modal Models are guarded state machines where at each state there is an associated actor, known as a refinement, that is fired when the model is in that state.

Although this characterization of control-oriented behavior is very general, it stands in contrast with Dataflow models where the behavior of a system is structured in terms of the movement of data rather than control. Counterexamples then include Kahn Process Networks [11], Dataflow as described by Lee and Matsikoudis [12], and Stream models such as those of Broy et al. [6]. However, it can be the case that a control-oriented model can participate in a heterogeneous system where a dataflow actor is internally control-oriented or a control-oriented model contains control locations at which a dataflow process represents the associated action taken, as can be the case in Modal Models for instance. Additionally, it must be emphasized that the monotonicity discussed in this paper is not the same as that of stream functions addressed in papers such as [6].

When these control-oriented models represent isolated models of computation, the formal treatment of their meaning in terms of operational semantics provides a clear way to reason about them, and gives a way to determine how to correctly implement them. However, particularly in the context of heterogeneous models, it is difficult to reason about compositional properties of these models given there is no clear general way to compose operational semantics such as those given by Boussinot and de Simone for Esterel [4], Berry for Constructive Esterel [3], and Andre for SyncCharts [1]. Since these languages are all both control-oriented and synchronous, a motivating kind of heterogeneity arises when these languages are decomposed into control-oriented fragments in synchronous compositions, as well as in systems with dataflow components, as was mentioned above.

In the case of SyncCharts, these two components are the control-oriented *State Transition Graphs*, which are similar to StateCharts [10], and synchronous compositions of *Macrostates* [1]. While in the operational semantics given by Andre [1] these two components are entangled, a denotational semantics would allow each of these parts to be treated separately, and the full model to arise out of their heterogeneous composition. Such a denotational formalism for synchronous composition exists in the Synchronous Reactive (SR) model of computation given by Edwards [8]. In this model, the semantics of a step in execution is given by the least fixed point of the function derived from composing the functional representations of each component in the model. So long as these components can be represented as monotonic functions, this least fixed point is guaranteed to uniquely exist.

Using the SR model to express synchronous composition, one should be able to achieve a model similar to that of SyncCharts or Constructive Esterel as a heterogeneous, hierarchical composition of control-oriented models and SR models. But, reasoning about this composition requires a general denotational semantics for control-oriented models. In particular, with this kind of semantics the conditions can be determined under which such a model is monotonic. Having such a denotational semantics for control-oriented models facilitates the analysis of other compositional properties of these models as well. Like the SR model, there are other models of computation that can similarly be described in a compositional way, as is done by Tripakis et al. [16]. With the semantics given in this paper, meaningful compositions can be formed between control-oriented models and these other models.

1.1 Contributions

In order to reason about control-oriented models in a compositional manner, this paper presents the *Coroutine Model of Computation*, a general denotation formalism for control-oriented models. This model consists of atomic elements called *Continuation Actors*, and defines *Coroutine Models* as networks of these Continuation Actors. A Coroutine Model composes Continuation Actors to form itself a Continuation Actor. Taking influence from the idea of *stars* and *Reactive Cells* from Andre's SyncCharts [1,2], the decisions to take control transitions in Coroutine Models are treated as part of the individual Continuation Actors in the network. This choice avoids having to settle on a particular language and semantics for transition guards and actions, and leads to a simple compositional semantics for Coroutine Models, defined in terms of the behavior of their constituting Continuation Actors.

Moreover, a meaning is given to non-strict Continuation Actors, which can make partial control decisions given partial inputs. Correspondingly, a non-strict dynamic semantics is defined for Coroutine Models containing these non-strict Continuation Actors. Further, it is argued that these semantics, in fact, form monotonic functions when the constituting Continuation Actors of a model are monotonic. Thus, Coroutine Models can be meaningfully put into synchronous compositions such as the that of SR models. What we give here is an abstract semantics [13] for concurrent composition of sequential processes. Our semantics focuses on the control behavior, and hence complements a semantic that focuses on concurrency, such as SR [8] or KPN [11].

2 Continuation Actor

A *Continuation Actor* describes a process that has a set of programmatic *entry locations* starting from which execution can be *entered*, concluding by either *terminating*, *suspending*, or *exiting* with an *exit label*. A Continuation Actor *exiting* represents the control leaving the Continuation Actor and moving to some external location referred to by the *exit label*. A Continuation Actor *terminating* represents the end of control flow, whereas a continuation *suspending* denotes a pause taken in control flow, yielding control to a containing model or an execution environment. A suspended Continuation Actor can be *resumed*, which can be thought of as *entering* with a special, relative entry location that is set internally to the location at which the Continuation Actor was last suspended. Finally, continuations can be *initialized*, which too can be thought of as a special entry location.

2.1 Continuation Actor Semantics

Formally, a *Continuation Actor* **C** is defined by the tuple

$$\mathbf{C} = (\mathbb{I},\ \mathbb{O},\ \mathbb{S},\ s_0,\ \mathcal{L},\ \mathcal{G},\ \textbf{enter},\ \textbf{fire},\ \textbf{postfire}) \tag{1}$$

similar to an *Actor* in *Modular Actor Interface* semantics [16]. The first three types represent the input $\mathbb{I}$, output $\mathbb{O}$, and state $\mathbb{S}$ of the Continuation Actor, with the initial state $s_0 \in \mathbb{S}$. The subsequent two components, $\mathcal{L}$ and $\mathcal{G}$, are finite sets containing *entry locations* and *exit labels.* Together, these six components specify the static semantics of the Continuation Actor. With the addition of special elements to $\mathcal{L}$ and $\mathcal{G}$, entry and exit *control actions* for **C** are defined as follows

$$\mathbb{L} = \mathcal{L} + \mathit{initialize}_u + \mathit{resume}_u \tag{2}$$

$$\mathbb{G} = \mathcal{G} + \mathit{terminate}_u + \mathit{suspend}_u \tag{3}$$

where T_u is the singleton type containing T and $+$ is a disjoint union. Actions *initialize* and *resume* in $\mathbb{L}$ denote the initialization and resumption of a Continuation Actor, whereas actions in $\mathcal{L}$ denote entrance of a Continuation Actor at the corresponding location. Similarly, *terminate* and *suspend* denote the result of a Continuation Actor terminating and suspending. Actions in $\mathcal{G}$ denote exiting a Continuation Actor via the corresponding exit labels.

The last three components form the *interface* of a Continuation Actor, and define its dynamic semantics. They have the following types:

$$\textbf{enter} : \mathbb{S} \times \mathbb{I} \times \mathbb{L} \to \mathbb{G} \tag{4}$$

$$\textbf{fire} : \mathbb{S} \times \mathbb{I} \times \mathbb{L} \to \mathbb{O} \tag{5}$$

$$\textbf{postfire} : \mathbb{S} \times \mathbb{I} \times \mathbb{L} \to \mathbb{S} \tag{6}$$

The **fire** and **postfire** function are similar to those in [16] and [14], only differing in their additional input of an entry action. The **fire** function specifies the outputs produced by the Continuation Actor with the given state, input, and entry action. The **postfire** likewise specifies the change in state consequent the execution from a given entry. The **enter** function specifies the control behavior of the Continuation Actor, and is the extension of the interface beyond that of an *Actor* [16]. In particular, this function specifies the concluding *control decision* made by the execution in the form of an exit action.

The role these interface functions play in execution depends on the model of computation in which the Continuation Actor is contained. In the case of an SR model, for instance, a typical execution is constituted of a series of discrete steps. In each step n, there will be several iterations, indexed by k, computing a least fixed point of the relation determined by the contained elements. A Continuation Actor would, in a particular state $s_n \in \mathbb{S}$, be *entered* and *fired* for each iteration with a particular entry action l_n^k and input value i_n^k, producing a exit action g_n^k and output value o_n^k. The Continuation Actor would then be *postfired* at the end of the step updating its internal state from s_n to s_{n+1} in terms of the final values for the input and entry action, denoted i_n^M and l_n^M. Such an execution would fulfill the relations

$$g_n^k = \mathbf{enter}(s_n, i_n^k, l_n^k) \tag{7}$$

$$o_n^k = \mathbf{fire}(s_n, i_n^k, l_n^k) \tag{8}$$

$$s_{n+1} = \mathbf{postfire}(s_n, i_n^M, l_n^M) \tag{9}$$

Note that the state is not superscripted by an iterative step since it is maintained over iterations of a fixed point computation. Later, the semantics of the particular case of a Coroutine Model will be described in detail.

2.2 Non-strict Continuation Actors

In certain models of computation, input and output values can be partially known during execution. Examples of this include Synchronous Reactive models [8] and models in synchronous languages such as Constructive Esterel [3] and SyncCharts [1]. The input and output types of components in these models are extended to represent this partial information by being lifted into pointed Complete Partial Orders (pCPOs), which are partially ordered sets containing a bottom element $\perp$ and the least upper bound of each chain.

In the case where partial information simply means that a variable may either be known to have a particular value or not known, the corresponding pCPO often used is constructed by adjoining a bottom element to the set of values associated with the type of the variable. In this case, all particular values are incomparable in the order, and all greater than the adjoined bottom element. This pCPO is known as a flat CPO. For tuples of variables, which often characterize input and output spaces, the corresponding pCPOs are typically the pointwise products of the flat CPOs for each constituting variable. Nevertheless, for generality it is not assumed that any of these particular pCPOs is used.

Given that the spaces $\mathbb{I}$ and $\mathbb{O}$ are lifted into pCPOs a Continuation Actor can be specified on these lifted types. Consequently, the **fire** function can be defined so that partial information about the outputs can be determined from partial information about the inputs. A function is known as *strict* if it maps all input values that are not maximal in the input pCPO to bottom. A function is otherwise *non-strict*, and intuitively can be understood as able to determine some information about the output without total information about the input. Non-strict functions play an important role in models of computation such SR [8] where constructive methods are used to iteratively determine consistent valuations of input and output variables which can have cyclic dependencies. Note that the state here is not lifted into a pCPO, and thus the **postfire** function has no non-strict version analogous to that of **fire**. The **enter** function can similarly extended to operate over partial information about inputs producing partial information about exit actions in $\mathbb{G}$. These partial control choices can be represented as sets of possible exit actions given the partial information about the input. Hence the **enter** function in such a Continuation Actor has a lifted codomain of type $2^{\mathbb{G}}$.

If this representation of partial information about control actions, $2^{\mathbb{G}}$, is ordered by reverse-inclusion, where

$$a \leq b \;=\; a \supseteq b$$

a pCPO is formed with $\bot$ being the whole set $\mathbb{G}$. The motivation behind this ordering is that a strict increase in this order corresponds to making more specific control decisions, with the singleton elements representing a unique and thus total decision. Monotonicity of the **enter** function, as required in certain domains with constructive semantics, therefore corresponds to the requirement that

$$\forall\, s \in \mathbb{S},\, l \in \mathcal{L} \bullet a \leq b \;\Rightarrow\; \textbf{enter}(s,\, a,\, l) \supseteq \textbf{enter}(s,\, b,\, l)$$

Intuitively, this monotonicity property means that as more is known about the input, the control choices at the least do not become less known, and may become more known. In particular, for the **enter** function this means that as more is known about the input additional control choices can never be added, and some may be removed potentially narrowing down the control behavior to a single choice.

2.3 Counter Example

An example of a Continuation Actor is a **Counter** that increments an internal state s_c each time it is resumed, and subsequently suspends. This **Counter** also has a threshold stored in an internal state s_t, which can be set by an input i_t. If the **Counter** is resumed and $s_c \geq s_t$, instead of suspending the **Counter** exits with exit label g_t. Let this **Counter** also have an output o_c that is set to the current count during each execution. In order to set the value of the threshold s_t to input i_t, suppose there is also an explicit entry location l_t at which s_t is set before performing the resume action. Assume that i_t, s_c, s_t, and o_c are all natural numbers (of type $\mathbb{N}$).

This **Counter** can defined formally as follows. Let the static semantics be

$$\textbf{Counter} \;=\; (\underbrace{\overbrace{\mathbb{N}}^{\mathbb{I}}}_{i_t},\, \underbrace{\overbrace{\mathbb{N}}^{\mathbb{O}}}_{o_c},\, \underbrace{\overbrace{\mathcal{L} \times \mathbb{N} \times \mathbb{N}}^{\mathbb{S}}}_{(s_l,\, s_c,\, s_t)},\, \overbrace{(l_0,\, 0,\, 0)}^{s_0},\, \overbrace{\{l_t,\, l_0,\, l_1\}}^{\mathcal{L}},\, \overbrace{\{g_t\}}^{\mathcal{G}})$$

The state here is a triple $(s_l,\, s_c,\, s_t) \in \mathcal{L} \times \mathbb{N} \times \mathbb{N}$, where $s_l \in \mathcal{L}$ holds the entry location to resume at after a suspension, $s_c \in \mathbb{N}$ is the current counter value, and $s_t \in \mathbb{N}$ is the current threshold value. In addition to the entry location l_t, which sets the threshold, there are two internal entry locations l_0 and l_1. s_l is set to l_0 initially, and in this state the counter is reset under a resumption, but upon the completion of any entry s_l is set by the **postfire** function to l_1, indicating that the **Counter** is counting when it is resumed. The interface functions are then

$$\textbf{enter}((s_l,\, s_c,\, s_t),\, i_t,\, l) \;=\; \begin{cases} \mathit{suspend} & l = l_0 \text{ or } \mathit{initialize} \\ \textbf{enter}((s_l,\, s_c,\, s_t),\, i_t,\, s_l) & l = \mathit{resume} \\ \textbf{if } s_c \geq s_t \textbf{ then } g_t \textbf{ else } \mathit{suspend} & l = l_1 \\ \textbf{enter}((s_l,\, s_c,\, i_t),\, i_t,\, l_r) & l = l_t \end{cases}$$

$$\textbf{fire}((s_l,\, s_c,\, s_t),\, i_t,\, l) \;=\; \begin{cases} 0 & l = l_0 \text{ or } \mathit{initialize} \\ \textbf{fire}((s_l,\, s_c,\, s_t),\, i,\, s_l) & l = \mathit{resume} \\ s_c + 1 & l = l_1 \text{ or } l_t \end{cases}$$

$$\textbf{postfire}((s_l,\, s_c,\, s_t),\, i_t,\, l) \;=\; \begin{cases} (l_1,\, 0,\, s_t) & l = l_0 \text{ or } \mathit{initialize} \\ \textbf{postfire}((s_l,\, s_c,\, s_t),\, i_t,\, s_l) & l = \mathit{resume} \\ (l_1,\, s_c + 1,\, s_t) & l = l_1 \\ \textbf{postfire}((s_l,\, s_c,\, i_t),\, i_t,\, l_r) & l = l_t \end{cases}$$

Note that here there is a difference between internal location l_1 and entry action *resume*, and likewise between l_0 and *initialize*, and that these cannot be conflated. If the **Counter** were entered with *resume* in its initial state, it would be map to l_0 rather than l_1. Although *initialize* is always the same case as l_0, *initialize* is maintained as separate as a matter of satisfying the interface obligation of providing such an entry action.

2.4 State Example

Another example of a Continuation Actor would be one that represents a state in a state machine, where the state evaluates outgoing guard expressions as part of its **enter** function and performs corresponding transition actions as part of its **fire** function. Let this formulation of a state, called **StateCA**, be parameterized by a finite set of transitions $\mathcal{T}$ and a default action $q_{\mathbf{def}}$. Each transition $\tau_k \in \mathcal{T}$ is defined by the tuple $\tau_k = (p_k,\, q_k,\, g_k)$, where $p_k : \mathbb{I} \to 2$ are transition predicates, $q_k : \mathbb{I} \to \mathbb{O}$ are transition actions ($q_{\mathbf{def}}$ is of the same type), and g_k are exit labels, referring to the remote destination of control upon taking the corresponding transition. Let π_p, π_q, and π_g be the projection functions for these components.

Given these parameters, characterizing the local behavior of the state, such a **StateCA** A can be given the following static semantics:

$$A(\mathcal{T},\, q_{\mathbf{def}}) = (\mathbb{I},\, \mathbb{O},\, \overbrace{\mathbf{1}}^{\mathbb{S}},\, \overbrace{\mathbf{u}}^{s_0},\, \overbrace{\emptyset}^{\mathcal{L}},\, \overbrace{\{\pi_g(\tau) \,|\, \tau \in \mathcal{T}\}}^{\mathcal{G}})$$

Here, there are no explicit entry locations and the exit labels for A are the locations $\pi_g(\tau_k)$ corresponding to each transition τ_k. There is only one state, denoted $\mathbf{u}$. In addition to the given transitions let the set of transitions be adjoined with an additional default transition defined

$$\tau_{\mathbf{def}} = (\forall\, \tau \in \mathcal{T} \bullet \neg\pi_p(\tau),\, q_{\mathbf{def}},\, \mathit{suspend})$$

to form $\mathcal{T}'$. This predicate of this default transition is true if those of all other transitions are false, the action is the given default action, and instead of an exit label the third component denotes suspension. Assume that there also exists a function $\mathbf{choose}_{\mathcal{T}'}$ of type $\mathbb{I} \to \mathcal{T}'$ that, given an input, chooses a transition τ for which the predicate $\pi_p(\tau)(i)$ is true.[1] If none are true, let it return default transition. The interface functions for A are simply:

$$\begin{aligned}
\mathbf{enter}(s,\, i,\, l) &= \pi_g(\mathbf{choose}_{\mathcal{T}'}(i)) \\
\mathbf{fire}(s,\, i,\, l) &= \pi_q(\mathbf{choose}_{\mathcal{T}'}(i))(i) \\
\mathbf{postfire}(s,\, i,\, l) &= \mathbf{u}
\end{aligned}$$

3 The Coroutine Model of Computation

Models in the *Coroutine* Model of Computation describe networks of *Continuation Actors*, connected to each other such that the exit labels of one Continuation Actor refer to entry locations of another. The referent can either be an explicit entry location, the *initialize* action, or the *resume* action on a target Continuation Actor. Given this structure, when one Continuation Actor in the network *exits*, control can proceed to another Continuation Actor following these connections. Furthermore, when a Continuation Actor *suspends* or *terminates* during execution the containing Coroutine Model does as well. An execution of a Coroutine Model is thus a sequence of executions of the contained Continuation Actors forming a control path through the structure and terminating with either suspension or termination. The Coroutine Model is also given its own entry locations and exit labels that can connect internally to the respective exit labels and entry locations of its contained Continuation Actors.

In this manner, a Coroutine Model can also be *entered* by entering one of its entry locations, as well as be *resumed* by resuming the Continuation Actor in which the execution of the model had been previously suspended. The model can also be *initialized* by initializing a particular initial Continuation Actor. It can *exit* with one of its exit labels, and also *suspend* or *terminate* if one of its contained Continuation Actors does. It follows that a Coroutine Model is itself a Continuation Actor. This compositionality property allows for Coroutine Models to form hierarchies, and likewise for specified Continuation Actors to be built out of other Continuation Actors.

3.1 Coroutine Models

Formally, a *Coroutine Model* $\mathcal{M}$ is described by the following tuple

$$\mathcal{M} = (\mathbf{Q},\, q_0,\, m_{\mathbb{I}},\, m_{\mathbb{O}},\, \oplus,\, \kappa,\, \eta) \tag{10}$$

Here, $\mathbf{Q}$ is a finite set of Continuation Actors that constitute the model, and $q_0 \in \mathbf{Q}$ is an initial Continuation Actor. The two components $m_{\mathbb{I}}$ and $m_{\mathbb{O}}$ map

[1] This allows for the possibility that the predicates are not mutually exclusive in which case **choose** determines a means to select a unique transition.

between the inputs and outputs of the whole model and those specific inputs and outputs of particular Continuation Actors in $\mathbf{Q}$.

Let $\mathbb{I}_{\mathcal{M}}$ and $\mathbb{O}_{\mathcal{M}}$ be the input and output types of $\mathcal{M}$. The input and output maps then have the following types:

$$m_{\mathbb{I}} : \Pi\ q \in \mathbf{Q} \bullet \mathbb{I}_{\mathcal{M}} \rightarrow \mathbb{I}_q \tag{11}$$

$$m_{\mathbb{O}} : \Pi\ q \in \mathbf{Q} \bullet \mathbb{O}_q \rightarrow \mathbb{O}_{\mathcal{M}} \tag{12}$$

where the operator Π here denotes a dependent type product, and the types $\mathbb{I}_q$ and $\mathbb{O}_q$ denote the input and output types for Continuation Actor q. By composition with $m_{\mathbb{I}}$ and $m_{\mathbb{O}}$, the input and output types of each Continuation Actor are made identical. The binary operator $\oplus : \mathbb{O}_{\mathcal{M}} \times \mathbb{O}_{\mathcal{M}} \rightarrow \mathbb{O}_{\mathcal{M}}$ is then used to combine the mapped output values produced by different Continuation Actors.

Let two sets of internal entry locations and exit labels be defined for the model

$$\mathcal{L} = \Sigma\ q \in \mathbf{Q} \bullet \mathbb{L}_q \tag{13}$$

$$\mathcal{G} = \Sigma\ q \in \mathbf{Q} \bullet \mathcal{G}_q \tag{14}$$

where the operator Σ here denotes a dependent type sum. In other words, members of $\mathcal{L}$ are of the form $(q,\ x)$ where $q \in \mathbf{Q}$ and $x \in \mathbb{L}_q$, and likewise for $\mathcal{G}$ with respect to $\mathcal{G}_q$. Let the finite sets $\mathcal{L}_{\mathcal{M}}$ and $\mathcal{G}_{\mathcal{M}}$ be the entry locations and exit labels of the model, distinct from their internal counterparts.

The functions κ and η give the structure to the model. The former maps locations in $\mathcal{L}_{\mathcal{M}}$ to internal locations in $\mathcal{L}$. The latter maps each exit labels of each Continuation Actor to either entry actions of another, including to the *initialize* and *resume* special locations, or to exit labels in $\mathcal{G}_{\mathcal{M}}$. They can therefore be given the following types:

$$\kappa : \mathcal{L}_{\mathcal{M}} \rightarrow \mathcal{L} \tag{15}$$

$$\eta : \mathcal{G} \rightarrow \mathcal{L} + \mathcal{G}_{\mathcal{M}} \tag{16}$$

When the conclusion of the execution of q is to exit with exit label g, and $(q',\ k) = \eta(q,\ g)$, control proceeds with entry action k performed on Continuation Actor q'. When instead $\eta(q,\ g) \in \mathcal{G}_{\mathcal{M}}$, control exits the model.

The state space of model $\mathcal{M}$ is constructed from a product of the state spaces of the Continuation Actors in $\mathbf{Q}$ along with the internal entry location corresponding to the entry action to be taken when the model resumes from a suspension. That is

$$\mathbb{S}_{\mathcal{M}} = \mathcal{L} \times \prod_{q \in \mathbf{Q}} \mathbb{S}_q \tag{17}$$

Correspondingly, the initial state of $\mathcal{M}$ is

$$s_{0\mathcal{M}} = ((q_0,\ \textit{initialize}),\ s_{0\,q_1},\ \ldots,\ s_{0\,q_n}),\ \text{ where } q_k \in \mathbf{Q}, 1 \leq k \leq n \tag{18}$$

so that calling *resume* on model in its initial state has the effect of initializing q_0.

The above, in total, give the static semantics for Coroutine Model $\mathcal{M}$ as a Continuation Actor:

$$\mathbf{C}_{\mathcal{M}} = (\mathbb{I}_{\mathcal{M}},\, \mathbb{O}_{\mathcal{M}},\, \mathbb{S}_{\mathcal{M}},\, s_{0\mathcal{M}},\, \mathcal{L}_{\mathcal{M}},\, \mathcal{G}_{\mathcal{M}}) \tag{19}$$

3.2 Strict Dynamic Semantics

For a *Coroutine Model* $\mathcal{M}$, specified as in (19), **enter**, **fire**, and **postfire** functions can be defined compositionally, in terms of the specifications and corresponding interfaces of the contained Continuation Actors in **Q**. The definitions of these functions constitute a denotational dynamic semantics for Coroutine Models as Continuation Actors.

In order to describe the traversal of control in the model, the interface functions are augmented using the input and output maps to functions that have types corresponding to the model:

$$\mathbf{enter}_U(s,\, i,\, (q,\, l)) \;=\; (q,\, \mathbf{enter}_q(s_q,\, m_{\mathbb{I}}(q,\, i),\, l)) \tag{20}$$

$$\mathbf{fire}_U(s,\, i,\, (q,\, l)) \;=\; m_{\mathbb{O}}(q,\, \mathbf{fire}_q(s_q,\, m_{\mathbb{I}}(q,\, i),\, l)) \tag{21}$$

$$\mathbf{postfire}_U(s,\, i,\, (q,\, l)) \;=\; r_q(s,\, \mathbf{postfire}_q(s_q,\, m_{\mathbb{I}}(q,\, i),\, l)) \tag{22}$$

where the function $r_q(s,\, v)$ replaces the element in s corresponding to q with value v.

The process of traversing a control path through the model, following exit labels of Continuation Actors to entry locations of subsequent Continuation Actors, ultimately reaching suspension, termination, or the exiting of the model, is described by the $\mathbf{enter}_U$ function in conjunction with the structural map η. In order to put these two pieces together, first it should be noted that the type of $\mathbf{enter}_U$ function is

$$\mathbf{enter}_U : \mathbb{S}_{\mathcal{M}} \times \mathbb{I}_{\mathcal{M}} \times \mathcal{L} \to \mathcal{G} + \mathcal{Z} \tag{23}$$
$$\text{where } \mathcal{Z} \;=\; Q \times (\mathit{suspend}_u + \mathit{terminate}_u)$$

When the image of $\mathbf{enter}_U$ is in $\mathcal{G}$, control then can continue to another Continuation Actor determined by η, whereas if the image is in $\mathcal{Z}$ the control ends in the model with a suspension or termination. To connect this with η, an augmented version of the function is defined as follows:

$$\nu : \mathcal{G} + Q \times \{\mathit{terminate},\, \mathit{suspend}\} \to \mathcal{L} + \mathcal{G}_{\mathcal{M}} + \mathcal{Z} \tag{24}$$

$$\nu(g) = \begin{cases} \eta(g) & g \in \mathcal{G} \\ g & g \in \mathcal{Z} \end{cases} \tag{25}$$

This function can then be composed with $\mathbf{enter}_U$ to form the *traversal function*, which describes the control traversal through the model:

$$\epsilon : \mathbb{S} \times \mathbb{I}_{\mathcal{M}} \times \mathcal{L} \to \mathcal{L} + \mathcal{G}_{\mathcal{M}} + \mathcal{Z} \tag{26}$$

$$\epsilon(s,\, i) \;=\; \nu \circ \mathbf{enter}_U(s,\, i) \tag{27}$$

Since $\mathcal{L}$ is in both the domain and codomain of ϵ, it can be iterated over, starting with an initial location l_0, forming a series

$$(l_0,\, \epsilon(s,\, i)(l_0),\, \epsilon(s,\, i)^2(l_0),\, \ldots)$$

possibly ending with a terminating value in either $\mathcal{G}_{\mathcal{M}}$ or $\mathcal{Z}$. This series generated by ϵ is the *control path* of the model for state s and input i, generated by location l_0.[2]

In addition to defining the traversal function with η, the map κ can be augmented to handle the whole set of entry actions $\mathbb{L}_{\mathcal{M}}$. To this end, let this augmentation be defined

$$\theta : \mathbb{S}_{\mathcal{M}} \times \mathbb{L}_{\mathcal{M}} \to \mathcal{L} \tag{28}$$

$$\theta(s,\, h) = \begin{cases} (q_0,\, \mathit{initialize}) & h \in \mathit{initialize}_u \\ (s_{\mathcal{L}},\, \mathit{resume}) & h \in \mathit{resume}_u \\ \kappa(h) & h \in \mathcal{L}_{\mathcal{M}} \end{cases} \tag{29}$$

$$\mathbf{enter}(s,\, i,\, h) \;=\; \mathbf{e}(s,\, i,\, \theta(s,\, h)) \tag{30}$$

$$\mathbf{e}(s,\, i,\, l) \;=\; \begin{cases} z, \text{ where } (q,\, z) = l & l \in \mathcal{Z} \\ l & l \in \mathcal{G}_{\mathcal{M}} \\ \mathbf{e}(s,\, i,\, \epsilon(s,\, i,\, l)) & l \in \mathcal{L} \end{cases} \tag{31}$$

$$\mathbf{fire}(s,\, i,\, h) \;=\; \mathbf{f}(s,\, i,\, \theta(s,\, h),\, 0_{\oplus}) \tag{32}$$

$$\mathbf{f}(s,\, i,\, l,\, o) \;=\; \begin{cases} o & l \in \mathcal{Z} + \mathcal{G}_{\mathcal{M}} \\ \mathbf{f}(s,\, i,\, \epsilon(s,\, i,\, l),\, o \oplus \mathbf{fire}_U(s,\, i,\, l)) & l \in \mathcal{L} \end{cases} \tag{33}$$

$$\mathbf{postfire}(s,\, i,\, h) \;=\; \mathbf{p}(s,\, i,\, \theta(s,\, h)) \tag{34}$$

$$\mathbf{p}(s,\, i,\, l) \;=\; \begin{cases} r_{\mathcal{L}}(s,\, l) & l \in \mathcal{Z} \\ r_{\mathcal{L}}(s,\, (q_0,\, \mathit{initialize})) & l \in \mathcal{G}_{\mathcal{M}} \\ \mathbf{p}(\mathbf{postfire}_U(s,\, i,\, l),\, i,\, \epsilon(s,\, i,\, l)) & l \in \mathcal{L} \end{cases} \tag{35}$$

Fig. 1. Strict dynamic semantics for Coroutine Models

The **enter**, **fire**, and **postfire** functions for a coroutine model can then be recursively defined as in figure 1, where the functions **e**, **f**, and **p** are the recursive kernels of the respective **enter**, **fire**, and **postfire**. Here, the **enter** function simply follows the control path of the traversal function. The **fire** function makes

[2] If this path has no terminating value, it is possible that iterating ϵ can diverge. Depending on the context, this can either be left as a possibility or restricted in some fashion as for instance is done in Esterel **loop** constructs in [3].

the same traversal, but accumulates outputs from the firing of each Continuation Actor with $\oplus$. The **postfire** function similarly traverse the control path, updating the state of each Continuation Actor along the path.

It should be noted that the state update over the traversal is independent from the traversal itself and the firings, hence it represents a set of changes that are only committed to after the traversal and output values are established. Nevertheless, if the model is suspended and resumed, the state changes take effect when it is resumed. Amongst these state changes is certainly, in particular, the change in the location at which to resume.

3.3 Non-strict Semantics

In a Coroutine Model where the constituting Continuation Actors are defined over pCPOs, representing partial information about inputs, outputs, and control decisions, a non-strict semantics can be given. Rather than determining a single control path through the Continuation Actors, given partial input information several control actions may be possible at each Continuation Actor, thereby generating instead a *tree* of control paths. If the **enter** function of each Continuation Actor is monotonic, then as the input information becomes more specific the control choices at each Continuation Actor in the tree become no greater, and possibly fewer, thereby pruning the *control tree* (or at least making it no larger).

Given the **enter** function for a Continuation Actor defined over pCPOs has a codomain of $2^{\mathcal{G}_q}$, the correspondingly lifted version of $\mathbf{enter}_U$ is defined:

$$\mathbf{enter}_U(s,\, i,\, (q,\, l)) \;=\; \{(q,\, g) \,|\, g \in \mathbf{enter}_q(s_q,\, m_{\mathbb{I}}(q,\, i),\, l)\} \tag{36}$$

The $\mathbf{fire}_U$ function on the other hand has essentially the same definition. Given this change in $\mathbf{enter}_U$, the function η can also be lifted:

$$\tilde{\nu}(G) \;=\; \{\eta(g) \,|\, g \in G \cap \mathcal{G}\} \cup (G \cap (\mathcal{G}_{\mathcal{M}} + \mathcal{Z}_{\mathcal{M}})) \tag{37}$$

Combining these two parts, a non-strict traversal function $\hat{\epsilon}$ can then be defined

$$\hat{\epsilon} : \mathbb{S} \times \mathbb{I}_{\mathcal{M}} \times \mathcal{L} \to 2^{\mathcal{L}+\mathcal{G}_{\mathcal{M}}+\mathcal{Z}} \tag{38}$$

$$\hat{\epsilon}(s,\, i) \;=\; \tilde{\nu} \circ \mathbf{e}(s,\, i) \tag{39}$$

Hence, for a given state and input, $\hat{\epsilon}$ maps a location to a set of successor locations potentially including terminal locations in $\mathcal{G}_{\mathcal{M}}$ or $\mathcal{Z}$. A control tree is thereby generated. If iterated over, $\hat{\epsilon}$ generates a tree of entry locations with terminations or suspensions as leaves. It is worth noting that the codomain of $\hat{\epsilon}$ can also be expressed as $2^{\mathcal{L}} \times 2^{\mathcal{G}_{\mathcal{M}}} \times 2^{\mathcal{Z}}$, and thus the image of $\hat{\epsilon}$ can always be decomposed into these three sets denoting the possible control choices within each of their respective categories.

Non-strict versions of the **enter** and **fire** functions for the coroutine model can then be defined in terms of their kernels $\mathbf{e}$ and $\mathbf{f}$ as in figure 2. The form

$$\mathbf{e}(s,\, i,\, l) \;=\; \begin{cases} \{z\},\ \text{where } (q,\, z) = l & l \in \mathcal{Z} \\ \{l\} & l \in \mathcal{G}_{\mathcal{M}} \\ \displaystyle\bigcup_{l' \in \hat{e}(s,\, i,\, l)} \mathbf{e}(s,\, i,\, l') & l \in \mathcal{L} \end{cases} \tag{40}$$

$$\mathbf{f}(s,\, i,\, l,\, o) \;=\; \begin{cases} o & l \in \mathcal{Z} + \mathcal{G}_{\mathcal{M}} \\ \displaystyle\prod_{l' \in \hat{e}(s,\, i,\, l)} \mathbf{f}(s,\, i,\, l',\, o \oplus \mathbf{fire}_U(s,\, i,\, l)) & l \in \mathcal{L} \end{cases} \tag{41}$$

Fig. 2. Nonstrict dynamic semantics for Coroutine Models

of both definitions is similar, in that the control tree in each is traversed recursively building a collection of results for all control paths. These results are then combined by an operation to form the greatest consistent conclusion that can drawn across all of them. In the non-strict **enter**, the sets of final control decisions for each path are combined with a union to conservatively give a set of all reachable control decisions for the model. In the **fire** function, an output is computed along each path, and the outputs for all paths are combined with a greatest lower bound. The resulting partial output consists of only the consistent information across all possible paths.

Given this non-strict characterization of Coroutine Model semantics, Coroutine Models can be given a clear denotational meaning in the context of fixed-point semantics. Synchronous compositions of Coroutine Models can therefore be constructed within semantics such as those of SR [8]. Important in such synchronous models is the property of monotonicity, which can be reasoned about in a clear way with the above denotational semantics. In order to perform this kind of domain-theoretic reasoning about the above semantic equations it must be determined that these equations have clear domain-theoretic solutions. In fact, this is the case, and the following can be proven:

Theorem 1. *Given a non-strict Coroutine Model $\mathcal{M}$, if the input $\mathbb{I}_{\mathcal{M}}$ and output $\mathbb{O}_{\mathcal{M}}$ types of the model are finite-height pCPOs and operator $\oplus$ is monotonic, then the above recursive equations characterizing the kernel functions* **e** *and* **f** *have unique least fixed-point solutions in the partial order of functions with codomains $2^{\mathbb{G}}$ and $\mathbb{O}_M$, respectively.*

Since solutions to the equations for the recursive kernels exist, then it can further be asked if under the right conditions **enter** and **fire** are monotonic functions from $\mathbb{I}_{\mathcal{M}}$ to $2^{\mathbb{G}}$ and $\mathbb{O}_M$, respectively. In fact these functions are monotonic, and more specifically continuous, under the conditions described in the following theorem:

Theorem 2. *Given a non-strict Coroutine Model $\mathcal{M}$, if the input $\mathbb{I}_{\mathcal{M}}$ and output $\mathbb{O}_{\mathcal{M}}$ types of the model are finite-height pCPOs and operator $\oplus$ is monotonic,* *and if for each $q \in Q$ the functions* $\mathbf{enter}_q$ *and* $\mathbf{fire}_q$ *are monotonic in terms of* $\mathbb{I}_q$*, and the mapping functions* $m_{\mathbb{I}}$ *and* $m_{\mathbb{O}}$ *are monotonic, then the non-strict kernels* $\mathbf{e}$ *and* $\mathbf{f}$ *are continuous in terms of* $\mathbb{I}_{\mathcal{M}}$*.*

It follows that **enter** and **fire** defined in terms of these non-strict kernels are both continuous, and thus monotonic as well. The proof of this fact involves noting that the union operator is the greatest lower bound under the order of reverse inclusion. Both definitions then are formally similar and can be altered in simple ways to get the same general formula for both. This general formula, taking the greatest lower bound of every branch formed by the traversal, can be shown to be monotonic because an increase in the codomain of the traversal function, ordered by reverse inclusion, corresponds to there being fewer branches, and thus fewer possible control paths. Furthermore, the greatest lower bound of a set of decreasing size always corresponds to the value of this bound remaining equal or increasing. That is

$$A \sqsubseteq B \;\Leftrightarrow\; A \supseteq B \;\Rightarrow\; \bigsqcap A \sqsubseteq \bigsqcap B \tag{42}$$

The proof follows from working out the details of this general relationship. The most important consequence of this theorem is that, under the above conditions, monotonicity is compositional for Coroutine Models.[3]

4 Related Work

The semantics of the component-based model *42* defined by Maraninchi [15] also gives an atomic interface description for *components* that includes control along with data, but the aims of the control dimension are very different. The control ports of a component in *42* receive tokens from a model *controller*, whereas the entry locations and exit labels of Continuation Actors are meant to form a network of control relationships. Since the control behavior of a *42* model is specified by its *controller*, which can be any imperative program, *42* by itself does not constitute any particular dynamic semantics. Moreover, there is no particular way in the interface semantics of *42* to handle non-strict control decisions.

A denotational semantics for Stateflow is given by Hamon [9] in which states are represented as continuations. However, the denotations given are functional programs relevant particularly in the context of understanding compilation. Since these functional programs act on both data and continuation environments, there is no clear way give this formalism a non-strict interpretation or compose it with other models that do not involve its environments. Although Hamon's semantics provide a backtracking mechanism, partial information cannot be combined from several potential paths as it can in the non-strict semantics given here. Finally, Hamon's model treats transition guards and actions as a part of the semantics of the execution. Here, in contrast, the role of guards and

[3] The full proof of the above two theorems is available in the appendix of: `http://chess.eecs.berkeley.edu/pubs/902.html`.

transition actions are considered to be part of the Continuation Actors. The semantics of Coroutine Models is thereby considerably simpler and applicable to a wider set of cases where different guard languages or other mechanisms are used to determine control transitions.

Another denotational formalism for state machines is given by Broy, et al. [7] as part of a general denotational semantics for UML. Although the denotational nature of this formalism lends itself to reasoning about composition, the formalism requires a complex state mechanism involving a frame stack and an event-driven model of behavior. The state machine semantics given does not provide a mean to articulate sequences of immediate transitions within a computational step, as opposed to transitions that happen in separate steps. Moreover, no investigation is made in this work of how to deal with non-strict state machines. One should note that the monotonicity properties discussed by Broy, et al., are not with respect to the state machines themselves being non-strict, in the manner this paper discusses, but instead regarding the causal properties of the event passing system cast into the formulation of streams.

5 Conclusion

The *Coroutine Model of Computation* defined here provides a general denotational model for representing control-oriented behavior, capable of use in hierarchical and heterogeneous systems. Both a strict and a non-strict denotational semantics have been given for Coroutine models allowing the compositional analysis between these models, and models with other semantics, to be expressed in functional terms. In particular, the non-strict semantics enable such models to be used in synchronous compositions with clear conditions for monotonicity. Coroutine Models also fit the definition of a *Director* given by Tripakis et al.[16] as a function from the interfaces of the constituting actors and structure of the model to an actor representation of the composite model. Given this language, many control-oriented models can be expressed in its terms by defining a set of constituting Continuation Actors and potentially making small modifications to the model semantics. Some preliminary work has thus far been done for fully modeling the semantics of SyncCharts [1] in terms of Coroutine Models. Work has also been done to implement the Coroutine Model of Computation in the **Ptolemy II** environment, where it can be used to develop and test executable heterogeneous models.

References

1. André, C.: SyncCharts: A visual representation of reactive behaviors. Rapport de recherche tr95-52, Université de Nice-Sophia Antipolis (1995)
2. André, C.: Computing synccharts reactions. Electronic Notes in Theoretical Computer Science 88(33), 3–19 (2004)
3. Berry, G.: The Constructive Semantics of Pure Esterel. In: Program (1999)
4. Boussinot, F., de Simone, R.: The ESTEREL language. Proceedings of the IEEE 79(9), 1293–1304 (1991)

5. Boussinot, F.: Reactive C: An extension of C to program reactive systems. Software: Practice and Experience 21(4), 401–428 (1991)
6. Broy, M.: Modelling Operating System Structures by Timed Stream Processing Functions. Journal of Functional Programming, 1–26 (1992)
7. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML, Towards a System Model for UML, Part 3: The State Machine Model. Technical report, Technische Universität München (2007)
8. Edwards, S.A.: The Specification and Execution of Heterogeneous Synchronous Reactive Systems. Technical report, University of California Berkeley (1997)
9. Hamon, G.: A denotational semantics for Stateflow. In: Proceedings of the 5th ACM International Conference, pp. 164–172 (2005)
10. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
11. Kahn, G., MacQueen, D., et al.: Coroutines and networks of parallel processes (1976)
12. Lee, E.A.: The semantics of dataflow with firing. Semantics to Computer Science 0720882(c), 1–20 (2008)
13. Lee, E.A., Neuendorffer, S.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits Systems 12(3), 231–260 (2003)
14. Lee, E.A., Tripakis, S.: Modal models in Ptolemy. In: 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT), vol. 47, pp. 11–21 (2010)
15. Maraninchi, F., Bouhadiba, T.: 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, pp. 53–62. ACM (2007)
16. Tripakis, S., Stergiou, C., Shaver, C., Lee, E.A.: A Modular Formal Semantics for Ptolemy. Mathematical Structures in Computer Science (to appear, 2012)