Illustrating the Coroutine Model of Computation

Helge Müller
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
hmu@informatik.uni-kiel.de

ABSTRACT

There are several tools in many programming languages for the Development of reactive realtime-systems and control-oriented models. But most of them are not usable since they have problems with modeling concurrency. Threading is not suitable for analyzing and executing control-oriented models, since it is difficult to ensure avoidance of simple concepts like *deadlocks*, *starving* or ensure *fixed order*. A more suitable method is *cooperative* or *partly cooperative scheduling*. The languages with *partly cooperative scheduling* are threaded programming languages, which use customized methods to implement cooperation. There is no uniform way of how these methods are implemented. Even in languages that use cooperative scheduling many different approaches are implemented. However, there should be a common ground to analyze different languages and tools. Shaver *et al* [4] use a Coroutine Model of Computation (Coroutine MoC). This paper delivers a short introduction to this Model of Computation (MoC) by giving an example and comparing it to Synchronous C (SC).

Keywords

SyncCharts, synchronous programming, coroutines, Model of Computation

1. INTRODUCTION

Tripakis *et al* [5] utilize *Actors*, as introduced in the Actor Model of concurrent Computation by Agha [1], to use them in Ptolemy. Ptolemy is a modeling and simulation software for concurrent realtime-systems. The Actor Model of Computation is using actors to achieve concurrent computation of programs which is similar to the computation of coroutines. Actors can also be used to determine monotonicity and continuity by observing these instances interacting with each other. The Coroutine MoC can be used to determine similar attributes for Coroutines. In the Coroutine MoC Shaver *et al* [4] use Continuation Actors (CA) to formally describe *Coroutines* and then use the Coroutine MoC to describe the interaction between these Continuation Actors.

1.1 Coroutines

Coroutines as introduced by Conway [2] are parts of *separable programs*. A *separable program* is a program that can be broken into parts, that fulfil requirements, to ensure that there are no unwanted side effects when splitting the program into multiple *Coroutines*. This means that a program is split into several parts that can be executed simultaneously with only a few synchronized points of communication. Therefore it is giving requirements to program parts to be used in a concurrent manner. As seen in Figure 1, the coroutines are regulating the control flow.

1.2 Synchronous C

To simulate cooperative scheduling similar to coroutines SC can be used. Synchronous C combines the widely distributed C programming language with the support for synchronous computation for *SyncCharts*. SyncCharts are used to visualize *reactive systems* and has a visual syntax, which enables it to have lucid programs. SC does not require specific tools other than the SC library and a C compiler. It uses special functions from its library to incorporate the synchronous features into C. These functions are used to direct the control flow in the program and for synchronous computation. Since SC uses cooperative scheduling, the threads have to cease control to the scheduler, to do this the *PAUSE* command can be used. This behavior is similar to the behavior of coroutines, when viewing the different threads in SC as coroutines. The SC code preserves the *SyncCharts* structure, so it can be used to visualize running code in a editor (e.g. Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)).

1.3 Coroutine Model of Computation

The Coroutine Model of Computation as described by Shaver *et al* [4] is a denotational formalism, which provides a common ground for *control-oriented models*. It has a similar approach as Tripakis *et al* [5] to provide a denotational language to describe *control-oriented behavior*. It provides strict as well as non-strict semantics. The Coroutine MoC includes *Control Tokens* to the *Actor Model* [5] turning the Actors into *Continuation Actors*. The Coroutine MoC uses *Control Tokens* to initiate, suspend, resume and terminate the itself as well as special labels that direct the control flow of the Coroutine MoC. These labels are used to transfer the control from one Continuation Actor (CA) to another Continuation Actor inside the Coroutine MoC. This behavior does reflect the behavior of cooperative scheduled Coroutines with each CA representing a separable part of a program (Coroutine).

The Coroutine Model is therefore a network of Continuation Actors which is similar to combining Coroutines to receive a program. The Continuation Actors in the network use *cooperative scheduling* to reduce chances of non-deterministic behaviour and *state explosion* [6].

Subject of this paper is to illustrate how the Coroutine MoC can be used and where problems might occur when working with the Coroutine Model of Computation. To achieve this, the paper contains a simple Coroutine MoC example with three CA, it also compares the general use of the Coroutine MoC to SC.

2. CONTINUATION ACTOR

Continuation Actors are a varied form of *Actors*. Actors consist of inputs, outputs and states. Furthermore a CA contains *entry locations* and *exit labels* as well as the capability of *terminating* and *suspending*. A CA that is *suspending* or *terminating* stops its execution. During suspension the control is given to either a containing model or the execution environment until it is *resumed*. A resuming Continuation Actor resumes to its last state as if this state is an *entry location*. It also consists of a special *initializing state* through which it can be *initialized*. Furthermore it contains an *initial state* which is the default state.

Shaver *et al* [4] extend the Actor A defined by Tripakis *et al* [5]

$$A = (\mathbb{I}, \mathbb{O}, \mathbb{S}, s_0, F, P, D, T) \tag{1}$$

to a Continuation Actor C replacing the timed actor specific parts D(deadline) and T(time-update) with the **enter** function which represents the concluding control decisions.

$$C = (\mathbb{I}, \mathbb{O}, \mathbb{S}, \mathcal{L}, \mathcal{G}, s_0, \mathbf{enter}, \mathbf{fire}, \mathbf{postfire}) \tag{2}$$

the first three parts represent sets of inputs ($\mathbb{I}$), outputs ($\mathbb{O}$) and states ($\mathbb{S}$) of the CA. $s_0 \in \mathbb{S}$ is the initial state. $\mathcal{L}$ contains a set of entry locations and $\mathcal{G}$ contains a set of exit labels. Since the exit labels used by the Continuation Actor when suspending or terminating are not part of $\mathcal{G}$, the corresponding actions are added to the set.

$$\mathbb{G} = \mathcal{G} + terminate_u + suspend_u \tag{3}$$

Equally the actions for initializing and resuming need to be added to the entry locations.

$$\mathbb{L} = \mathcal{L} + intialize_u + resume_u \tag{4}$$

The last three components of the Continuation Actor define the dynamic semantics of the Continuation Actor. The **enter** function has the type

$$\mathbf{enter} : \mathbb{S} \times \mathbb{I} \times \mathbb{L} \to \mathbb{G} \tag{5}$$

meaning that it uses a state, an input and an entry location to determine the exit label. Similar types are used for the **fire** and **postfire** functions.

$$\mathbf{fire} : \mathbb{S} \times \mathbb{I} \times \mathbb{L} \to \mathbb{O} \tag{6}$$

The **fire** function uses the same inputs as the **enter** function but it generates an output $o \in \mathbb{O}$. The **postfire** function

$$\mathbf{postfire} : \mathbb{S} \times \mathbb{I} \times \mathbb{L} \to \mathbb{S} \tag{7}$$

returns a state $s \in \mathbb{S}$. The exact role of the last three *interface* elements is defined by the MoC the Actor is used in. The $\mathbb{I}$ and $\mathbb{L}$ with the state $\mathbb{S}$ are defining the current state from which the **enter** function determines the exit label $\mathbb{G}$. The **fire** function determines the output $\mathbb{O}$ and the **postfire** functions determines the following state $\mathbb{S}$.

Figure 1: Cooperative coroutine threads: No scheduler is required and threads explicitly resume each other at specific synchronization points. Figure from Hanxleden [3].

2.1 Observer Example

Utilizing this model we can create an Observer similar to the one in the first example from Hanxleden [7]. It is primarily routing the input to the output.

$$C_{obs} = (\mathbb{N}, \mathbb{N}, \mathcal{L}, (l_i), \{l_0, l_i\}, \{g_i, g_c\}) \tag{8}$$

Since it is using numbers as values the domain of the inputs as well as the outputs are in $\mathbb{N}$. Figure 2.a shows a general Continuations Actor with the general labels for inputs, outputs , states , entry locations and exit labels. The only two possible states for the actor is uninitialized and running. The Actor is entered through the l_i entry location to initialize it. While running, it is entered through the l_0 entry location. Since it only has these two states the domain for the states is $\mathcal{L}$, as seen in figure 2. The initial state, here when not initialized, is s_0 which is reached through the entry location l_i. The exit label g_i is used for exiting from the initial state, this is used in a later example 3.2 to notify other actors to initialize themselves. g_c gives the control to the consumer in the example 3.2 and is used when the observer is running.

Also needed are the **enter**, **fire** and **postfire** functions. Since they are not included in C_{obs} the complete model is $C_{obs} + (\mathbf{enter}_{obs}, \mathbf{fire}_{obs}, \mathbf{postfire}_{obs})$, where the functions are defined as

$$\mathbf{enter}_{obs} = \mathbf{enter}_o((s_l), i, s_l) \tag{9}$$

$$\mathbf{fire}_{obs} = \mathbf{fire}_o((s_l), i, s_l) \tag{10}$$

$$\mathbf{postfire}_{obs} = (l_0) \tag{11}$$

Postfire, in this case, always results in the observer running, since it keeps running after being initialized. The other two functions,$\mathbf{enter}_o$ and $\mathbf{fire}_o$ have three different cases, being initialized, resumed and running.

$$\mathbf{enter}_{obs} = \begin{cases} g_i & l = l_i \text{ or } initialize \\ \mathbf{enter}_o(s_l, i, s_l) & l = resume \\ g_c & l = l_0 \end{cases}$$

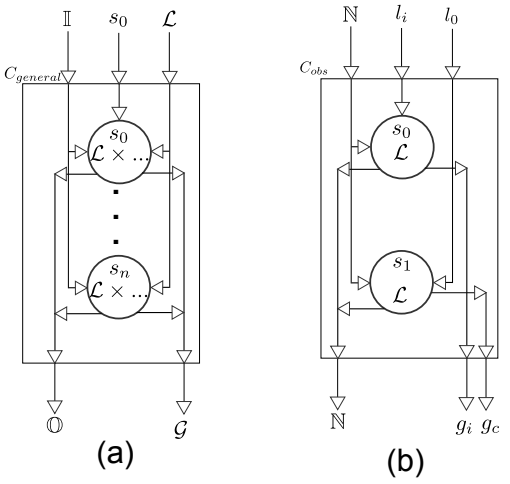

Figure 2: Continuation Actor and Observer example

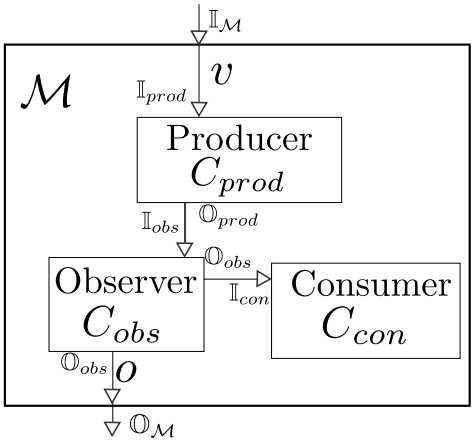

Figure 3: Inputs and Outputs of the PCO example

When running or initializing the **enter** function surrenders control through g_c and g_i respectively. This can be seen in Figure 2.b. Since s_0 can only be reached when initializing and the control then can only be ceased through g_i. Similar s_1 can only be reached when the CA is running and ceasing control will then occur through g_c. Otherwise, when the function resumes, it calls itself in the state previous to the suspension.

$$\mathbf{fire}_{obs} = \begin{cases} 0 & l = l_i \text{ or } initialize \\ \mathbf{fire}_o((s_l, i, s_l) & l = resume \\ i & l = l_0 \end{cases}$$

The $\mathbf{fire}_{obs}$ function uses 0 as initial value, this could also be i since $i = 0$ when initializing (in s_0), in this case 0 is used. When running (in s_1), it is returning the value i for output. And when resuming it is, similar to $\mathbf{enter}_{obs}$, resuming with the state from right before suspending.

This model can now be used in a Coroutine Model of Computation as *Continuation Actor* and will be referred to as C_{obs}.

3. COROUTINE MODEL OF COMPUTATION

3.1 Structure of the coroutine MoC

A *coroutine* MoC describes a network of *Continuation Actors*. It connects actors by binding exit labels to entry locations. Note that entry locations and exit labels are not containing the actions suspend, terminate, initialize and resume. These actions are bound to the corresponding actions of the model. Therefore the model mimics its actors when terminating or suspending. It also contains entry locations and exit labels, which connect to the entry locations and exit labels of the contained actors. Executing a Coroutine MoC is equivalent of executing its connected actors. Using this construct, the Coroutine MoC can be used similar to an actor, and therefore be entered, suspended, resumed and terminated. The Coroutine Model, similar to the Continuation Actor, can be formally described.

$$\mathcal{M} = (\mathbf{Q}, q_0, m_{\mathbb{I}}, m_{\mathbb{O}}, \oplus, \kappa, \eta) \tag{12}$$

With $\mathbf{Q}$ as the set of CA contained in the Coroutine MoC, the initial Continuation Actor q_0 as well as $m_{\mathbb{I}}$ and $m_{\mathbb{O}}$ to map inputs and outputs of the Coroutine Model to inputs and outputs of CA contained in $\mathbf{Q}$. Special attention has to be spend on the types, which are also equalized through the use of $m_{\mathbb{I}}$ and $m_{\mathbb{O}}$. $\oplus$ is a function used to combine two outputs to one ($\oplus : \mathbb{O} \times \mathbb{O} \to \mathbb{O}$). The two last components map the entry locations of the model to entry locations of the actors and exit labels of actors to either entry locations of other actors or to exit labels of the Coroutine MoC.

$$\kappa : \mathcal{L}_{\mathcal{M}} \to \mathcal{L} \tag{13}$$
$$\eta : \mathcal{G} \to \mathcal{L} + \mathcal{G}_{\mathcal{M}} \tag{14}$$

κ maps the entry locations of the model to entry locations of CA. η maps the exit labels of the CA to either entry locations of other actors or an exit label of the Coroutine MoC.

3.2 Example of a coroutine MoC

Utilizing this for the PCO example one possible implementation of the Coroutine MoC is using CA for the *Producer*, *Consumer* and *Observer*. The *Observer* was defined in section 2. The *Consumer* in this example is not saving more than one value. Saving more values could be implemented using the internal states of the Actor, but since it would complicate the internal states it is not done here. It is also not possible to rerun the model without restarting it. This, as well is done to condense and to simplify the example.

3.2.1 Producer Actor

The *producer's* task is to fill the buffer, this is achieved not by "saving" a value into a variable but by sending the value through the output of the producer, through the observer, to the input of the consumer. While the consumer is not really "consuming" the value, it is saving the last value in its internal states. The consumer is not producing the value as output since that behavior would render the observer useless. In a similar fashion to the CA used to define the Observer in section 2 it is now possible to define the producer and consumer.

$$C_{Prod} = (\mathbb{N}, \mathbb{N}, \mathcal{L} \times \mathbb{N} \times \mathbb{N}, (l_i, 0, 0), \{l_0, l_i\}, \{g_i, g_o, g_t\}) \quad (15)$$

The dynamic semantic of the producer actor contains the functions

$$\mathbf{enter}_{pr} = \mathbf{enter}_p((s_l, s_c, s_t), t, s_l)) \quad (16)$$
$$\mathbf{fire}_{pr} = \mathbf{fire}_p((s_l, s_c, s_t), t, s_l) \quad (17)$$
$$\mathbf{postfire}_{pr} = \mathbf{postfire}_p((s_l, s_c, s_t), t, s_l)) \quad (18)$$

Since the producer is the entity in the Coroutine MoC responsible for terminating the Coroutine MoC it is the only Continuation Actor that has more than the two exit labels for initializing and running, namely the termination label g_t. Furthermore it has an integrated counter similar to the counter example from Shaver [4], containing a threshold, to limit the values produced by the producer. The threshold is set when initializing the Model using the input and cannot be changed after initializing. Similar to the observer, the producers **enter** function consists of three cases.

$$\mathbf{enter}_{prod} = \begin{cases} g_i & l = l_i \text{ or } init \\ \mathbf{enter}_p((s_l, s_c, s_t), t, s_l) & l = resume \\ \mathbf{if}\ s_c > s_t\ \mathbf{then}\ g_t\ \mathbf{else}\ g_o & l = l_0 \end{cases}$$

In contrast to the observer, the producer consists of a if-then-else construct in the running case, which ensures the termination after reaching the threshold. Otherwise there is the resumption case that resumes the actor in the last state prior to suspension. The values $(s_l, s_c, s_t), t, s_l)$ represent the entry locations, the counter value and the threshold respectively.

$$\mathbf{fire}_{prod} = \begin{cases} 0 & l = l_i \text{ or } initialize \\ \mathbf{fire}_p((s_l, s_c, s_t), t, s_l) & l = resume \\ s_c & l = l_0 \end{cases}$$

The **fire** function also consists of three cases. The initial value, which could be freely chosen, is in this case set to 0. When running, the value is passed on, which, similar to the initial value, could be random, is in this case, similar to the example by Hanxleden, the increasing value s_c. The use of the **fire** function to send the value is due to not "saving" the value in a variable but using the output to transfer the value to the observer and ultimately the consumer.

$$\mathbf{postfire}_{prod} = \begin{cases} (l_0, 0, s_t) & l = l_i \text{ or } init \\ \mathbf{postfire}_p((s_l, s_c, s_t), t, s_l) & l = resume \\ (l_0, s_c + 1, s_t) & l = l_0 \end{cases}$$

When the **postfire** function is used to initialize the actor it simply returns the default state $(l_0, 0, s_t)$, with l_0 being the entry location for a running actor, the initial value of 0 for the counter and s_t being the threshold. When the actor is running, therefore entered through l_0, the **postfire** function is incrementing the counter value s_t every iteration by one. The entry location l_0 and the threshold s_t in the state are left as they are.

The *consumer* is constructed similar to the observer since its purpose is to receive the value from observer and save the last value received. To implement this and not making the observer obsolete by producing an output of the value as well, it does not need any outputs as seen in figure 3. Having no outputs is not prohibited in the Coroutine MoC but under normal circumstances it is not commonly occurring, since there is usually some kind of output expected. Furthermore the **fire** function is not needed since there are no outputs that need to be produced. If the implementation of the Coroutine MoC should require a **fire** function, a function returning a default value outside of the domain of the other output values could be an option.

$$C_{Con} = (\mathbb{N}, \emptyset, \mathcal{L} \times \mathbb{N}, (l_i, 0), \{l_0, l_i\}, \{g_p\}) \quad (19)$$

3.2.2 Consumer Actor

The *Consumer* has no exit label for initializing since it is the last entity that gets initialized and therefore the following entity is already initialized and needs to be entered through l_0. The default value is set to zero and again could be random. The domain for the output is $\emptyset$ since there are no outputs. The domain for the states is $\mathcal{L} \times \mathbb{N}$. The second value can be used to store the last received value.

$$\mathbf{enter}_{con} = \mathbf{enter}_c((s_l, s_v), i, s_l) \quad (20)$$
$$\mathbf{postfire}_{con} = \mathbf{postfire}_c((s_l, s_v), i, s_l) \quad (21)$$

The dynamic semantics of the consumer actor contain only these two functions, since the **fire** function is not needed and thereby it is ignored. As mentioned earlier this function could be a constant that is not in the domain of the outputs.

$$\mathbf{enter}_{con} = \begin{cases} g_p & l = l_i \text{ or } initialize \\ \mathbf{enter}_c((s_l, s_v), i, s_l) & l = resume \\ g_p & l = l_0 \end{cases}$$

Enter is not initialized with a default value of 0, instead it uses the same exit label as the running entry location, then giving control to the producer since it has "consumed" the

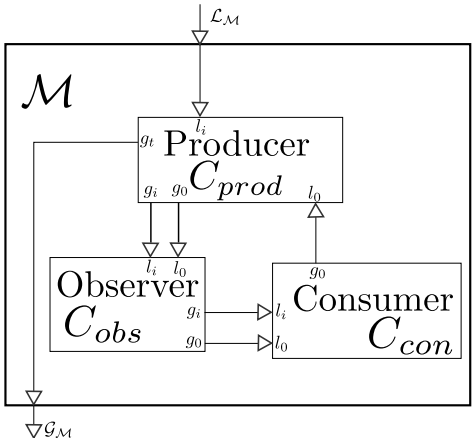

Figure 4: Controlflow of the PCO example

value. The resume case is constructed as they are for the other two actors.

$$\textbf{postfire}_{con} = \begin{cases} (l_0, 0) & l = l_i \text{ or } init \\ \textbf{postfire}_c((s_l, s_v), i, s_l) & l = resume \\ (l_0, i) & l = l_0 \end{cases}$$

The **postfire** function is initialized and resumed similar to the other two actors, using the default state and the last state before suspension respectively. When the actor is running it enters the state (l_0,i). At first the producer is initialized and therefore traverses control via the g_0 exit label to the l_i entry location. And so the control traverses through the model as previous described.

3.2.3 Coroutine Model of Computation Example

These are the actors for the model and therefore part of $\mathbf{Q}$ which itself is part of the models dynamic semantics $\mathcal{M}$.

$$\mathcal{M} = (\mathbf{Q}, q_0, m_\mathbb{I}, m_\mathbb{O}, \oplus, \kappa, \eta) \tag{22}$$

The model $\mathcal{M}$ also contains the initial actor q_0 in this case C_{Prod}. The mapping functions $m_\mathbb{I}$ and $m_\mathbb{O}$, which are used for mapping the inputs and outputs, unifying the types of mapped values. Therefore the mapping functions map the inputs from model to inputs from actors and outputs of actors to inputs of other actors or to the outputs of the Coroutine MoC. κ and η are mapping the exit labels and entry locations in a similar fashion to the inputs and outputs, giving the model its structure. The mapping of the entry location and exit labels can be seen in figure 4.

$$\begin{aligned}
\mathbf{Q} =& \{C_{Prod}, C_{Obs}, C_{Con}\} \\
q_0 =& (C_{Prod}) \\
m_\mathbb{I} =& (\mathbb{I}_\mathcal{M} \to \mathbb{I}_{C_{prod}}, \mathbb{I}_{C_{obs}} \to \mathbb{O}_{C_{prod}}, \mathbb{I}_{C_{con}} \to \mathbb{O}_{C_{obs}}) \\
m_\mathbb{O} =& (\mathbb{O}_{C_{prod}} \to \mathbb{O}_\mathcal{M}, \mathbb{O}_{C_{prod}} \to \mathbb{I}_{C_{obs}}, \mathbb{O}_{C_{obs}} \to \mathbb{I}_{C_{con}}) \\
\kappa =& (\mathcal{L}_\mathcal{M} \to C_{prod} l_i) \\
\eta =& (C_{prod} g_t \to \mathcal{G}_\mathcal{M}, C_{prod} g_i \to C_{obs} l_i, C_{prod} g_o \to C_{obs} l_0, \\
& C_{obs} g_i \to C_{con} l_i, C_{obs} g_c \to C_{con} l_0, C_{con} g_p \to C_{prod} l_0)
\end{aligned}$$

The models actors and dynamic semantics are thereby defined so the next step is to define its static semantics similar to the semantics of a CA

$$C_\mathcal{M} = (\mathbb{I}_\mathcal{M}, \mathbb{O}_\mathcal{M}, \mathbb{S}_\mathcal{M}, s_{0\mathcal{M}}, \mathcal{L}_\mathcal{M}, \mathcal{G}_\mathcal{M}) \tag{23}$$

$\mathbb{I}_\mathcal{M}$ being the inputs of the Coroutine MoC, in this example just one value $v \in \mathbb{N}$. The output $\mathbb{O}_\mathcal{M}$ also contains one value $o \in \mathbb{N}$ in this example. The states $\mathbb{S}_\mathcal{M}$ are defined through the product of all the states in the contained actors.

$$\mathbb{S}_\mathcal{M} = \mathcal{L} \times \prod_{q \in \mathbf{Q}} \mathbb{S}_q \tag{24}$$

$s_{0\mathcal{M}}$ is the initial state of the model, which is also initializing q_0 when it is *resumed*. $\mathcal{L}_\mathcal{M}$ represents the models entry locations and $\mathcal{G}_\mathcal{M}$ represents the exit labels of the model. Applying this onto the example we receive

$$\begin{aligned}
\mathbb{I}_\mathcal{M} =& \{v\} \\
\mathbb{O}_\mathcal{M} =& \{o\} \\
s_{0\mathcal{M}} =& ((q_0, init), s_{0_{C_{prod}}}(l_i, 0, 0), s_{0_{C_{obs}}}(l_i), s_{0_{C_{con}}}(l_i, 0)) \\
\mathcal{L}_\mathcal{M} =& \{\mathcal{L}_\mathcal{M}, C_{prod} l_i, C_{prod} l_0, C_{obs} l_i, C_{obs} l_0, C_{con} l_i, C_{con} l_0\} \\
\mathcal{G}_\mathcal{M} =& \{\mathcal{G}_\mathcal{M}, C_{prod} g_i, C_{prod} g_o, C_{prod} g_t, C_{obs} g_i, C_{obs} g_c, C_{con} g_p\}
\end{aligned}$$

The functions responsible for the traversal of control, namely **enter**, **fire** and **postfire** are defined as a combination of the **enter**, **fire** and **postfire** functions of the actors. To get these functions the **enter** function is combined with the maps $m_\mathbb{I}$ and $m_\mathbb{I}$ to get corresponding input and output types. After that the $\textbf{enter}_\mathcal{M}$ function is defined as

$$\textbf{enter}_\mathcal{M} : \mathbb{S}_\mathcal{M} \times \mathbb{I}_\mathcal{M} \times \mathcal{L} \to \mathcal{G} + \mathcal{Z} \tag{25}$$

$$\text{where } \mathcal{Z} = \mathbf{Q} \times (suspend_\mathcal{M} + terminate_\mathcal{M}) \tag{26}$$

If the image of $\textbf{enter}_\mathcal{M}$ is in $\mathcal{G}$ the control traverses to another actor, but if the image lies in $\mathcal{Z}$ the model terminates or suspends. To decide to which actor the control traverses to it uses the η function.

$$v : \mathcal{G} + \mathbf{Q} \times \{terminate, suspend\} \to \mathcal{L} + \mathcal{G}_\mathcal{M} + \mathcal{Z} \tag{27}$$

$$v(g) = \begin{cases} \eta(g) & g \in \mathcal{G} \\ g & g \in \mathcal{Z} \end{cases} \tag{28}$$

to get the traversal function it is now composed with the

$\mathbf{enter}_{\mathcal{M}}$ function

$$\epsilon : \mathbb{S} \times \mathbb{I}_{\mathcal{M}} \times \mathcal{L} \to \mathcal{L} + \mathcal{G}_{\mathcal{M}} + \mathcal{Z} \tag{29}$$

$$\epsilon(s,i) = v \circ \mathbf{enter}_{\mathcal{M}}(s,i) \tag{30}$$

To handle the entry locations a similar augmentation is done for κ.

$$\theta : \mathbb{S}_{\mathcal{M}} \times \mathbb{L}_{\mathcal{M}} \to \mathcal{L} \tag{31}$$

$$\theta(s,h) = \begin{cases} (q_0, initialize) & h \in initialize_{\mathcal{M}} \\ (s_{\mathcal{L}}, resume) & h \in resume_{\mathcal{M}} \\ \kappa(h) & h \in \mathcal{L}_{\mathcal{M}} \end{cases} \tag{32}$$

Utilizing these functions the $\mathbf{enter}_{\mathcal{M}}$,$\mathbf{fire}_{\mathcal{M}}$ and $\mathbf{postfire}_{\mathcal{M}}$ functions can be defined.

$$\mathbf{enter}(s,i,h) = \mathbf{e}(s,i,\theta(s,h)) \tag{33}$$

$$\mathbf{e}(s,i,l) = \begin{cases} z, \text{where } (q,z) = l & l \in \mathcal{Z} \\ l & l \in \mathcal{G}_{\mathcal{M}} \\ \mathbf{e}(s,i,\epsilon(s,i,l)) & l \in \mathcal{L} \end{cases} \tag{34}$$

$$\mathbf{fire}(s,i,h) = \mathbf{f}(s,i,\theta(s,h),0_{\oplus}) \tag{35}$$

$$\mathbf{f}(s,i,l,o) = \begin{cases} o & l \in \mathcal{Z} + \mathcal{G}_{\mathcal{M}} \\ \mathbf{f}(s,i,\epsilon(s,i,l), & \\ \quad o \oplus \mathbf{fire}_{\mathcal{M}}(s,i,l)) & l \in \mathcal{G}_{\mathcal{M}} \end{cases} \tag{36}$$

$$\mathbf{postfire}(s,i,h) = \mathbf{p}(s,i,\theta(s,h)) \tag{37}$$

$$\mathbf{p}(s,i,l) = \begin{cases} r_{\mathcal{L}}(s,l) & l \in \mathcal{Z} \\ r_{\mathcal{L}}(s,(q_0,init)) & l \in \mathcal{G}_{\mathcal{M}} \\ \mathbf{p}(\mathbf{postfire}_{\mathcal{M}}(s,i,l),i, & \\ \quad \epsilon(s,i,l)) & l \in \mathcal{G}_{\mathcal{M}} \end{cases} \tag{38}$$

Therefore the example is complete and can be used as a Coroutine Model of Computation.

4. COMPARING SC AND CMOC

To compare SC and Coroutine MoC one has to examine the similarities. Both are tools able to design SynchCharts and other synchronous systems. They both use multiple instances, which communicate with each other, are cooperative scheduling the control flow and are running concurrently. The Coroutine MoC uses CA and SC uses Threads as instances. However SC has a lot more structure in its design to handle scheduling whereas Coroutine MoC uses its already existing structure to naturally schedule its Actors. Coroutine MoCs can not be run since they are models and as far as i know no programs for their execution have been developed. So the design of a model is at times more theoretical than the implementation of a Synchronous C program. The theoretical approach is quite useful because it forces a thought through and well designed model, whereas a SC program could be just written down without proper thought. The Coroutine MoC is a mathematical model not bound to a programming language. In contrast SC is bound to C as a programming language.

5. CONCLUSION

The Coroutine MoC is in my opinion a well rounded model to describe and design coroutines and receive the expected semantics. Therefore it allows for a good design for continuation actors as well as whole coroutine models. Since it is a mathematical construct it can be hard to foresee results directly in the design(e.g. testing if the desired results are achieved). This is mostly due to a lack of implemented programs that work with this MoC. It is relatively easy to design a continuation actor and design the interactions between multiple actors. However in places the consequences of changing parts of a function, as mentioned earlier, can be extensive and hard to overlook. Combined with a lack of testing ability, it is quite difficult to get started. In contrast to SC which you download and learn the syntax and are able to construct a simple example easily. Otherwise when using the model it is quite clear what steps need to be done to complete the model since every model consists of the same components.

Coroutine MoC and SC can benefit from each other since they are working with a very similar concept, so it should be possible to implement the Coroutine MoC similar to the threads from SC or even using them. The communication between the threads is already in place, and so is the scheduling, so the only thing missing are internal states, exit labels and entry locations. Also there should be a mechanic that combines the single CA to the Coroutine MoC, combining the enter, fire and postfire functions and giving access to the inputs and outputs of the Coroutine MoC.

6. REFERENCES

[1] G. A. Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. page 12 ff, 1985.

[2] M. E. Conway. Design of a Separable Transition-Diagram Compiler. pages 1–3, 1963.

[3] C. Motika, R. von Hanxleden, and M. Heinold. Synchronous Java: Light-weight, Deterministic Concurrency and Preemption in Java. pages 1–2.

[4] C. Shaver and E. A. Lee. The Coroutine Model of Computation. 2012.

[5] S. Tripakis, C. Stergiou, C. Shaver, and E. A. Lee. A Modular Formal Semantics for Ptolemy. page 13 ff.

[6] A. Valmari. The state explosion problem. *Lecture Notes in Computer Science*, pages 429–437, 1998.

[7] R. von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, pages 225–234, Grenoble, France, Oct. 2009. ACM.