Embedded System Components using the Cal Actor Language

Johan Eker
Department of Automatic Control
Lund University
Box 118, 222 00 Lund, Sweden
johane@control.lth.se

Jörn W. Janneck
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720-1770, USA
janneck@eecs.berkeley.edu

ABSTRACT

Actors are computational entities that communicate with other actors and with the environment by passing tokens via their input and output ports. Actors are connected to form models or applications. An actor or a network thereof may be view as components. The semantics of a network of actor is determined not only by the functional behavior of the actors or how the actors are interconnected, but also on the model of computation. A proper component model allows for hierarchical composition of possible concurrent components. The behavior of the resulting component should be deterministic in both temporal and functional aspects. It is highly desirable to be able to investigate properties, such as bounded memory and absence of deadlock for components. One key for doing this is to have full knowledge of the communications patterns of the actors/component. Another issue for embedded actor components is the problem of generating code suitable for environments where the computing resources are scarce. One key to achieve this to have a good understanding of the control flow and memory management of a component. Cal is novel actor language designed to facilitate the extraction of needed properties in order handle the above issues. It is a small domain specific language for implementation of actors. Hierarchical components are designed by source level transformations of actor networks, resulting in single, semantically equivalent, actors, who in turn may serve as a component and be composed in new networks with other actors (components).

1. INTRODUCTION

Design of embedded software applications is inherently difficult and error prone due to their concurrent nature. A typical embedded application consists of several parallel activities which interacts with both each other and the external environment under timing constraints.

Embedded software is commonly implemented in a rather pedestrian fashion using inadequate development tools, requiring unnecessary many man-hours and a deep knowledge of low-level programming. While ten years ago the issues pertaining to embedded software design were the headache for a chosen few, today we find embedded software development in a vast number of industries. Embedded applications are present in consumer electronics, household equipment, cars, etc., and the ability to construct high quality embedded software with shorter lead than the competitors may be the difference between success and failure.

This paper addresses the issues of component technologies for embedded software through the use of a novel programming language called Cal - Cal Actor Language.

The paper is organized as follows. Section 1.1 introduces the Cal actor language, and this is followed by a discussion on the state of the of embedded systems components in Section 1.2. Section 1.3 discusses the notion of *Models of Computation* and gives a number of examples of how this may affect the behavior of a given actor. Section 2 gives an overview of Cal and Section 3 shows how Cal actors are composed. Finally, Section 4 discusses the design rationale behind Cal.

1.1 Cal Actor Language

The Cal actor language is a small, domain specific language designed to support the implementation of actors and the analysis of networks of such actors. It was created as a part of the Ptolemy project at UC Berkeley [12]. The concept of actors was first introduced in [6] as a means of modeling distributed knowledge-based algorithms. Actors have since then become widely used, for example see [2].

A Cal actor is an computational entity with input ports, output ports, states and parameters. It communicates with other actors by sending and receiving *tokens* along unidirectional connections. A model, or application, then consists of a network of interconnected actors. When an actor is executed it is said to be *fired*. During a firing, tokens on the input ports are consumed and tokens on the output ports are produced.

Cal is not intended as a full fledged programming language, but rather a small language to be embedded in an environ-

ment which provides the necessary infrastructure. It designed as a means to coordinate scheduling and communication in a component-based framework. The functionality of an actor is defined by a set of *actions* and their associated firing rules. The firing rules are conditions on the presence of tokens on the input ports and possibly also on their values. The Cal approach is in part much inspired by the work presented in [10], where actors are formally defined using firing functions and firing rules.

Cal facilitates the use of several techniques for checking compatibility between connected actor. Production and consumption rates for a actor may be extracted from a Cal actor and can, for example, be used to statically check a synchronous data flow model [8] that is built using Cal actors. A more powerful way to analyze actor compatibility is to use automata descriptions of the actor interfaces and analyze the behavior of the composite. One promising technique for doing this is interface automaton [3], where actor compatibility might be determined by matching possible legal input and output token sequences from the actors.

The goal with Cal is to provide a concise high-level description of an actor. As a side effect, we will insulate the actor behavior from the specificities of the APIs of different runtime platforms. The language itself does not specify a strict semantics, but instead leaves part of it to the designer of the particular platform or application in which the actor is executed. For example, the design of an add actor for a synchronous data flow model or for a Kahn [7] process network model should be identical. The control of the run-time behavior such as scheduling and data transfer is left outside the language.

1.2 Embedded Software Components

While state-of-the-art software technologies such as object oriented programming (C++/Java) and component frameworks (ActiveX/Java Beans/Corba) have been highly successful in desktop and web applications, they have been found less useful in an embedded setting. Component technologies allow programmers to package and reuse code in a standardized way. The functionality of a component, which usually is distributed in binary form, may be accessed using a well defined interface, which describes available methods and their signature. The information in the interface only describes the static type of the component and not its behavior. This approach falls short when it comes to handling temporal constraints and dynamic behavior, such as shared resources and efficient communications and scheduling. Consider, for example, the embedded controller application in Figure 1 a. The component consists of two parts that are executed synchronously. The first block is the controller and the second is a limiter. The component executes as a single thread with a period T, a deadline D and a worst case execution $WCET$. This component would be straightforward to deploy by simply connecting its input and output ports, and making sure that the component is schedulable. So far so good, but now assume that we would like to use this PID Component as a building block in the creation of a more complex controller component, see Figure 1 b. The Cascade PID Component consists of two PID components in series, executing asynchronously at different rates. It is desirable to view the Cascade PID as a being a component in the same sense as the PID. However, in order to deploy it we now longer has only three configuration parameters, but six. In order to correctly execute and schedule the Cascade Controller Component we must look inside it and treat it not as one but as two components. Hence, in this example,

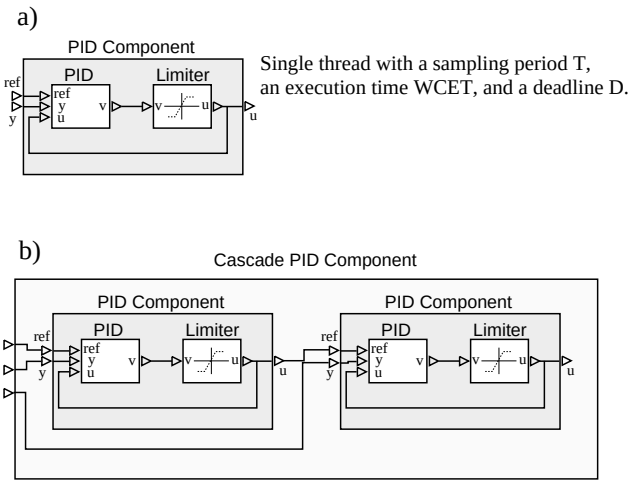


Figure 1: A controller component

encapsulation breaks and the component concept fails, and the reason for this is that threads just do not compose well. A good introduction to embedded software and a discussion of the shortcomings of current technologies and problems as the above are found in [11].

1.3 Dataflow Programming

The Ptolemy project [12, 5] at University of California at Berkeley addresses the problem of components for embedded systems in a more elegant way. A component here can be viewed as parametric in three orthogonal directions: functionality + scheduling + communication. The actual actor descriptions, e.g. the Cal code, capture the functional behavior, while the system handles the scheduling of the actors, the inter-actor communication and the communication with the environment.

In Ptolemy several actors may be composed into a *network*, a graph-like structure in which output ports of actors are connected to input ports of the same or other actors, indicating that tokens produced at those output ports are to be sent to the corresponding input ports. Such actor networks are of course key to the construction of complex systems and we can make the following observations:

- A connection between an output port and an input port can mean different things. It usually indicates that tokens produced by the former are sent to the latter, but there are a variety of ways in which this can happen: token sent to an input port may be queued in FIFO fashion, or new tokens may 'overwrite' older ones, or any other conceivable policy. It is important to stress that actors themselves are oblivious to these policies: from an actor's point of view, its input ports serve as abstractions of (prefixes of) input sequences

of tokens, while its output ports are the destinations of output sequences.

- Furthermore, the connection structure between the ports of actors does not explicitly specify the order in which actors are fired. This order (which may be partial, i.e actors may fire simultaneously), whether it is constructed at runtime or whether it can be computed from the actor network, and if and how it relates to the exchange of tokens among the actors—all these issues are part of the interpretation of the actor network.

The interpretation of a network of actors determines its *semantics*—it determines the result of the execution, as well as how this result is computed, by regulating the flow of data as well as the flow of control among the actors in the network. There are many possible ways to interpret a network of actors, and we call any specific interpretation a *model of computation* (MoC)–the Ptolemy project focuses on exploring the issues of models of computation and their composition.

To demonstrate how the same actor behaves differently depending on its MoC, consider the following two simple actor networks in Figures 2 and 3.

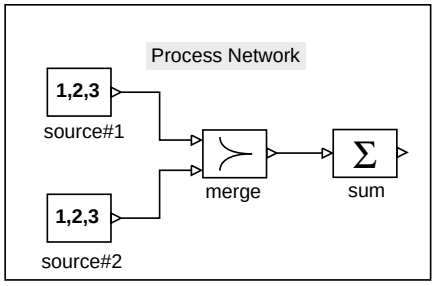


Figure 2: Process Network

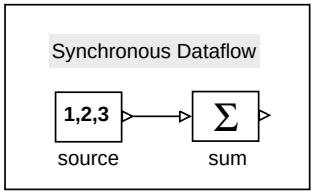


Figure 3: Synchronous Dataflow

The actors communicate by means of token passing. The system in Figure 2 consists of two source actors that produce tokens of type integer when *fired*. They are connected to a merge actor which transforms the two input token streams into one single token stream fed to the sum actor, which calculates the sum of all incoming tokens and outputs the result. The MoC is chosen as Kahn process networks (PN) which allows the actors to be treated as independent, unsynchronized threads, under the condition that all actors are prefix monotonic w.r.t their input token stream. Prefix monotonicity is achieved by making sure that the necessary number of tokens always are available at a port before trying to read from it (or as a blocking read). The implementation of the sum actor could look something like this:

```
actor Sum {
  Token sum = 0;

  void Fire() {
    //Runs as a thread

    while(true) {
    waitForToken();
    sum = sum + getToken();
    sendToken(sum);
  }
}
```

Now, consider the system in Figure 3 with one source actor and one sum actor. The MoC in this case is Synchronous Dataflow (SDF) [8], where the actors are executed synchronously and the schedule can be determined statically. Since the actor schedule is static, we can make sure that the actor is only fired when there are tokens available, relieving the actor of checking this. The implementation of the sum actor could now be simplified and might look something like this:

```
actor Sum {
  Token sum := 0;

  void Fire() {
    // Is externally invoked

   sum := sum + getToken();
    sendToken(sum);
  }
}
```

As a third example, consider an actor that will be used in fix point calculations. For example, an implementation where the actors are composed using synchronous language semantics [1] or in a simulation model where actors are composed using analog signal semantic, e.g. Matlab/Simulink [15]. In these compositions, actors must allow to be iterated in order to reach a fix point. In general, an actor must allow to be fired several times without updating its internal state. With this additional requirement, the sum actor would need to look something like this:

```
actor Sum {
  Token sum := 0; // The actor state
  Token $sum;     // Shadow of actor state

  void Fire() {
     // Is externally invoked
     // First make a temporary
     // copy of the state
     $sum := sum
     $sum := $sum + getToken();
```

```
    sendToken($sum);
  }

  void Commit() {
    sum := $sum;
  }
}
```

The `Fire`-method does not make any changes to the internal state and will thus allow repeated firing. Once a fix-point is reached, the `Commit`-method is called and the state is updated.

1.4 Actors as Components

A Cal application consists of a network of interconnected actors, scheduled according to a given MoC. The actual algorithm of the actor, i.e. the mapping of input tokens to output tokens, is completely separated from any execution specification, i.e. how it is scheduled and how and with whom it is communicating. The incomplete specification is an important feature of Cal and provides good ground for code reuse. The interface of an actor specifies a set of typed input and output ports, and the algorithm of the actor is defined as a set of input-output mappings called *actions*. Each action has an interface which describes firing conditions and token consumption and production. The interface of an actor does not only specify a *static* interface as found in object-oriented programming languages as Java, C++ etc, but also expresses constraints on how the interface is used. This can be thought of as defining a *dynamic* interface. A network of Cal actors may be transformed into a single Cal actor. An actor is a component and a network of actors is hence also a component. This allows for hierarchical system designs. An actor is only fully specified when placed in an environment, i.e. given a MoC. An actor implementation is hence refined upon composition, when the actor is interpreted in the context of a MoC. This greatly enhances code reuse, since algorithms, if implemented as actors, do not have to bother with runtime issues (scheduling etc), but those are left to the composer of a particular MoC.

2. CAL ACTORS

This section will give an introduction to the Cal actor language. The focus of the presentation will be on the actor interfaces and later it is shown how they can be used for compositions. Cal is part functional and part imperative. For example, the language has functional and procedural closure, side-effect free expressions, limited mutability of variables etc. This paper on discusses the part of Cal that we feel is relevant to the issues of composition. For a more comprehensive overview of the language see [4]. A few simple Cal examples are given below. We will start with a simple sum actor, which takes two numbers, adds them together and outputs the sum. The actor will have two input ports and one output port. It will operate on two, possibly infinite, input streams S_A and S_B

$$S_A = a_0, a_1, \ldots, a_n$$
$$S_B = b_0, b_1, \ldots, b_n$$

and produce a token streams C,

$$S_C = c_0, c_1, \ldots, c_n$$

which will have the following value

$$S_C = a_0 + b_0, a_1 + b_1, \ldots, a_n + b_n$$

When the actor *fires*, it consumes tokens on the input stream and produces a token on the output stream. There needs to be at least one token available at both S_A and S_B in order for the actor to be firable, and this is then the firing rule of the actor. This actor implemented in Cal as shown below.

EXAMPLE 1 (ACTOR). *A sum actor with two input ports A and B and one output port C. The type of all three ports are Double.*

```
actor Sum() Double A, Double B ⟹ Double C :
  Double sum := 0;
  action [a],[b] ⟹ [sum] do
    c := a + b;
  end
end
```

2.1 Actions

An action is an atomic piece of computation that an actor performs, usually in response to some input. The definition of an action describes three things:

- the *consumption* of input tokens,
- the *production* of output tokens,
- the *change of state* of the actor.

Usually, an actor definition contains a number of action definitions. Whenever it is fired the actor needs to choose one of them, and it does so based on the availability of input tokens, and possibly based on further conditions on their values, and its own state. The head of an action contains a description of the kind of inputs this action applies to, as well as the output it produces. The body of the action is a sequence of statements, that can change the state, or compute values for local variables that can be used inside the output port expressions.

Patterns and expressions are associated with ports either by position or by name. In the actor signature in Example 1, $Double\ A,\ Double\ B \Longrightarrow Double\ C$, an input pattern may look like '$[a]$, $[b]$' which binds a to the first token coming in on A and b to the first one from B. The same pattern may also be expressed using the port names: '$A :: [a]\ B :: [b]$'. This is often convenient if the actor has many input and output ports and it becomes cumbersome to associate the patterns and the ports using position.

2.2 Action matching

An actor can consist of any number of action definitions and when fired, it has to select one of them (or none, if none applies) for acting on the inputs and computing a new state and outputs. Firing an action will consume some tokens from the input sequences of the actor. For each action, a set of *patterns* describes how many tokens are consumed from each port if that action fires. In addition, such a pattern introduces a number of variables which are bound to the values of the respective tokens. An actor can only select an action that *matches* the current input in the current state. Such an action is said to be *firable*.

EXAMPLE 2 (ACTIONS). *The NondeterministicMerge actor reads tokens from either of its input ports and produces an output as a merge of its input streams. The first action can fire if there is at least one token on port A and similarly the second action can fire if there is at least one token on port B. However, if there are one or more tokens on both ports it is undecided which action that should be fired. Hence, in that respect this merge actor is nondeterministic.*

```
actor NondeterministicMerge()[T] T A, T B ⟹ T C :
  action [a], [] ⟹ [a] do end
  action [], [b] ⟹ [b] do end
end
```

The type of the ports is T, which is a type parameter. Type parameters are variable symbols that are bound to types when the actor is instantiated. They can be used to define type-relations between elements such as variables and ports inside the actor definition.

Input patterns allow a concise and intuitive description of input conditions, while at the same time facilitating a high degree of straightforward static analysis of properties such as:

- number of tokens consumed by an action,
- whether that number is constant, depending on parameters, or depending on the state,
- which channels are to be read from (in case of a multiport[1]),
- whether the patters are constant, depending on parameters, or varying with the state.

A common pattern is one that refers to the first few tokens in an input sequence, i.e. a pattern like [a, b, c]. This pattern introduces three new variables, and binds them to the first three tokens (from left to right) on the corresponding input port. Their type is the token type of that port.

EXAMPLE 3 (PORT PATTERNS). *Assume the input sequence* $[1, 2, 3, 4]$. *The pattern [a, b] matches, and binds a to 1, b to* 2.

The pattern [a, b | c] also matches, and binds a to 1, *b to* 2, *and c to* $[3, 4]$.

The pattern [a, b, c, d, e] does not match.

The above patterns all cause a fixed and statically determined number of tokens to be read from each channel that they match against. Often, however, the number of tokens to be read by an action cannot be statically determined, and in fact may depend on actor parameters or even the actor state. Repeating patterns provide a way of expressing this.

EXAMPLE 4 (REPEAT PATTERN). *This actor up-samples an input stream by an integer factor by inserting tokens with value zero. The up-sample factor is given by the factor parameter. On each firing, this actor reads one token from the input port and produces factor tokens on the output port. All but one of these is a zero-valued token of the same type as the input. The remaining one is the token read from the input. The position of this remaining one is determined by the phase parameter.*

```
actor UpSample[T](Integer factor, Integer phase)
  T input ⟹ T output :
  action [a] ⟹ [b] repeat factor do
    b := [if i = phase then a else 0 end :
          for Integer i in integers(1, factor)]
  end
end
```

Here integers(a, b) is a function that returns a list if integers from a to b.

2.2.1 Guards

The input conditions for an action must not only be defined by the pattern, but also using a *guard expression*. A guard is a set of boolean expressions that may impose additional conditions on the values of the variables bound by action port pattern.

EXAMPLE 5 (GUARDS). *The Sort-actor has two actions and sorts the input tokens according to a given sorting criterion, i.e. the function f, which is a parameter*[2]. *If the value of the evaluation of f is true the first action is chosen otherwise the second.*

```
actor Sorter[T]([T ⟶ Boolean ] f)
      T input ⟹ T output1, T output2 :
  action [a] ⟹ [b], [] guard f(a) : end
  action [a] ⟹ [], [b] guard not f(a) : end
end
```

Here the unambiguity from having two identical port patterns are resolved by additional constraints given by the guards.

Similarly, we can now create a merge actor which is deterministic and alternates between its two input ports.

EXAMPLE 6 (FAIRMERGE).

```
actor FairMerge1()[T] T A, T B ⟹ T C :
  Integer i := 0,
  action [a], [] ⟹ [a] guard i = 0 do
    i := 1
  end
  action [], [b] ⟹ [b] guard i = 1 do
    i := 0
  end
end
```

The last element in the header of an actor are type constraints. These can be used to impose conditions on the type variables. If these conditions are not met, the behavior of the actor is undefined, i.e. the author may assume that these

[1] Ports in Cal may be multidimensional. A multiport consists of a set of channel for reading and writing tokens.

[2] The type of a function is written as $[argumentType \longrightarrow returnType]$

conditions are true. Type constraints may require types to be equal, or may require a type to be a subtype/supertype of another type.

EXAMPLE 7 (TYPE CONSTRAINTS). *The type constraints of the Add-actor state the requirement that the port types must be of Number or a subtype thereof.*

actor $Add()[T < Number]\ T\ A,\ T\ B \Longrightarrow T\ C$:
  action $[a], [b] \Longrightarrow [a + b]$ do end
end

2.3 Action schedules and priorities

It is often useful to constrain the behavior of a actor in terms of its legal action firing order or in terms of the individual importance of each action. There are two constructs in Cal to achieve this, schedule and priority. In the previously shown Cal actors actions have been anonymous, but that need not be the case. Example 8 below shows an actor with named or tagged actions.

The legal firing order of the actions can be formulated as a regular expression over the tags. For example, let's say we have an actor A with the following action tags: connect, send_to_B, send_to_C, and disconnect. Suppose A is some sort of communication actor and transmits data to other actors over a network. The first thing that need to happen when the actor is started, is that it must connect to the network. After a link has been established, it may start to send token and finally as a last step before stopping it must close the connection. This constraint can be described by the following regular expression:

$$(connect\ (send\_to\_A \mid send\_to\_B)^*\ disconnect)^*$$

This is called an *internal action schedule* and can contain all the usual operators found in regular expressions such as: *, +, |, ?.

Using the action schedule construct, a variant of the FairMerge actor in Example 6 is given below.

EXAMPLE 8 (ACTION SCHEDULE). *The schedule section constraints the legal firing sequences so that the actor alternates between reading tokens of the two ports.*

actor $FairMerge2()[T]\ T\ A,\ T\ B \Longrightarrow T\ C$:
  $a1$: action $[a], [] \Longrightarrow [a]$ do ...end
  $a2$: action $[], [b] \Longrightarrow [b]$ do ...end
  schedule
    $(a1\ a2)*\ \mid (a2\ a1)*$
  end
end

Another way of constraining the firing sequence is to use action priorities, which are specified as chains of orders among tags.

EXAMPLE 9 (ACTION PRIORITIES). *The actor Foo has two input ports and three actions. Since the port patterns are overlapping, a priority clause is used to resolve the ambiguity and create a deterministic actor. In the case were tokens are available on both input ports, all three ports pattern will be enabled but only a1 will be fired due to higher priority.*

actor $Foo()[T]\ T\ A,\ T\ B \Longrightarrow T\ C$:
  $a1$: action $[a], [b] \Longrightarrow [a, b]$ do ...end
  $a2$: action $[a], [] \Longrightarrow [a]$ do ...end
  $a3$: action $[], [b] \Longrightarrow [b]$ do ...end
  priority
    $a1 > a2$,
    $a1 > a3$
  end
end

The partial order that is the transitive hull of the specified priority inequalities is the *priority relation* $>$ of the actor. Of the set S of enabled actions, only an action a may fire such that there is no action $b \in S$ such that $b > a$. Schedule and priority constructs may also be combined as shown in Example 10. This example several actions demonstrates how several actions may share the same tag and that a tag may be hierarchical.

EXAMPLE 10 (HIERARCHICAL TAGS). *Actor Bar has three actions, which read tokens from two input ports. The desired behavior is that the actor should, if possible, always fire action a1. However, if there are not tokens available on both of the ports action a1 is not enable, then the second or third actions should be selected. The internal order between a2.b1 and a2.b2 is not important, however, what is important is that equal amounts of tokens are consumed from both ports. To achieve this we use an internal schedule which constraints the internal order of a2.b1 and a2.b2. The priority clause forces the action a1 to be selected over any of the actions in a2.*

actor $Bar()[T]\ T\ A,\ T\ B \Longrightarrow T\ C$:
  $a1$: action $[a], [b] \Longrightarrow [a, b]$ do ...end
  $a2.b1$: action $[a], [] \Longrightarrow [a]$ do ...end
  $a2.b2$: action $[], [b] \Longrightarrow [b]$ do ...end
  priority
    $a1 > a2$
  end
  schedule
    $(a1 \mid (a2.b1\ a2.b2) \mid (a2.b2\ a2.b1))*$
  end
end

Two of the actions in actor Bar is grouped together under the tag a2. In the schedule and priority clauses a2 refers to all actions who has that a2 its top level tag.

3. COMPOSITION

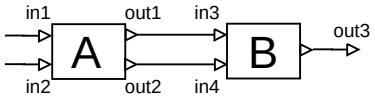
The Cal language specification is oblivious to the interpretation of an actor composite. The semantics of a network of interconnected Cal actors is left to the framework in which the actors are instantiated. The meaning of a Cal model does not only depend on the actors A and C, but also on the order in which they are executed (i.e. the schedule) and the way their communication is handled (i.e. buffering etc.).

First when we have chosen a MoC the semantics of the actor network is fully decided. Depending on the MoC we can make some static analysis of the Cal application and, for example, create schedules or detect deadlocks. The action patterns, the action priorities and the action schedules are three powerful concepts that allow control flow analysis and makes it possible to retrieve information about actor interaction. Below, we will give some examples of how deadlock can be detected and how a static schedule can be derived offline. Cal components are designed through source code level transformations, i.e. given a set of connected Cal actors and a MoC, a single, semantically equivalent, Cal actor is synthesized.

3.1 Data flow analysis

Checking that the data types of connected actors are correct, is only a first step in verifying that actors compatible. The next step is to look at the data flow through the actors through inspection of the input patterns and output expressions of the connected actors.

EXAMPLE 11 (DEADLOCK). *Consider the following model*



where the Cal code for the actors A and B is found below.

```
actor A[] () Double in1, Integer in2
                ⟹ Double out1, Integer out2 :
  action [a], [] ⟹ [], [d] do ... end
  action [], [b] ⟹ [], [d] do... end
end

actor B[] () Double in3, Integer in4 ⟹ Double out3 :
  action [a], [b] ⟹ [c] guard a > b do ... end
  action [a], [b] ⟹ [c] guard a ≤ b do ... end
end
```

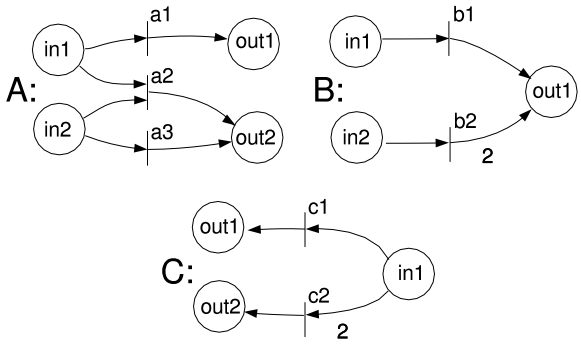
The data flow between actors can be analyzed by inspection of the connected output expressions and input patterns. In this trivial case it is straightforward to see that B will never produce any output, because there will never be an output from A that will match any of the actions of B. The model in the figure above will hence deadlock.

The example above demonstrated how the action signatures can be used to determine possible production and consumption rates for an actor. The general problem of using this information to detect deadlocks is undecidable, however we believe there are many cases where such analysis still would prove useful. The goal with Cal, in this setting, is to make it straightforward for analysis tools to extract the needed information from the actor source code.

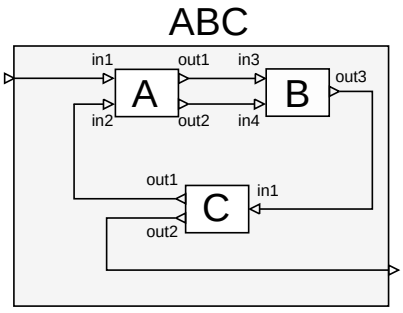
EXAMPLE 12 (COMPOSITION). *To demonstrate the ideas further, we will here give a more extensive example. Consider a system with the following three Cal actors:*

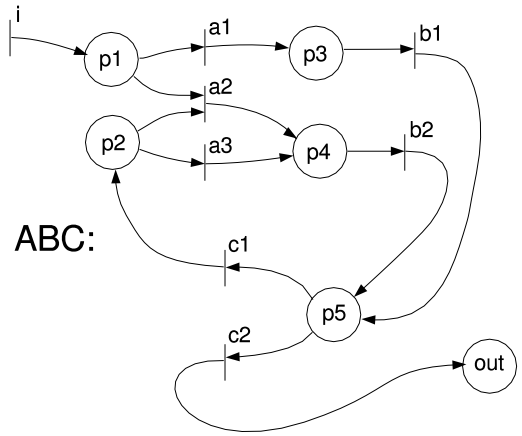
```
actor A[T] () T in1, T in2 ⟹ T out1, T out2 :
  a1 : action [a], [] ⟹ [c], [] do ... end
  a2 : action [a], [b] ⟹ [], [c] do ... end
  a3 : action [], [b] ⟹ [], [c] do ... end
  priority
    a2 > a1,
    a2 > a3,
    a1 > a3
  end
end
actor B[T] () T in1, T in2 ⟹ T out1 :
  b1 : action [a], [] ⟹ [c] do ... end
  b2 : action [], [b] ⟹ [c, d] do ... end
end
actor C[T] () T in1 ⟹ T out1, T out2 :
  c1 : action [a] ⟹ [a], [] do ... end
  c2 : action [a, b] ⟹ [], [a] do ... end
  schedule
    (c1 c2)*
  end
end
```

The production and consumption of tokens are modeled as petri net models, where places corresponds to inputs and output ports and transitions correspond to actions. Here it is assumed that they communicate with a discrete event model and that the buffer sizes are infinite.



When the above actors are composed to a model, the resulting petri net is given as the composite of the petri nets for the actors. This may then be used to investigate properties of the composite such as deadlock and memory bounds.





The incidence matrix for the above system is given as:

$$A = \begin{array}{c} \begin{matrix} P_1 & P_2 & P_3 & P_4 & P_5 \end{matrix} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 2 \\ 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -2 \end{bmatrix} \end{array} \begin{matrix} i \\ a1 \\ a2 \\ a3 \\ b1 \\ b2 \\ c1 \\ c2 \end{matrix}$$

The progression of the system is now described by

$$M = M_0 + A^T x,$$

where M is the resulting marking vector, M_0 the initial marking vector and x the firing vector. The firing rates for the actors in the composite can now be calculated from the nullspace of A^T, i.e. solving $A^T\mathbf{x} = 0$ w.r.t x. One solution in the example above is $\mathbf{x} = [1, 1, 0, 1, 1, 1, 1, 1]^T$, which corresponds to firing each of the actions in the composite once except a2 which is never fired. From $\mathbf{x}$ we can now derive a schedule:

$$S = [i,\ a1,\ b1,\ c1,\ a3,\ b2\ c2].$$

Given a static schedule it is now straightforward to calculate buffer size, i.e. maximum number of tokens at each place. Finally, we can make a program transformation and create the corresponding composite Cal actor:

```
actor ABC[T] () T in1 ⟹ T out1 :
  T $b1, $b2,          // The buffers
  ([⟶],[⟶]) A =(a1 : proc() begin ... end ,
                a3 : proc() begin ... end ),
  ([⟶],[⟶]) B =(b1 : proc() begin ... end ,
                b2 : proc() begin ... end ),
  ([⟶],[⟶]) C =(c1 : proc() begin ... end ,
                c2 : proc() begin ... end ),
  action [a] ⟹ [b] do
    $b1 := a;
    A.a1();     // Reads $b1, writes $b1
    B.b1();     // Reads $b1, writes $b1
    C.c1();     // Reads $b1, writes $b1
    A.a3();     // Reads $b1, writes $b1
    B.b2();     // Reads $b1, writes $b1, $b2
    C.c2();     // Reads $b1, $b2, writes $b1
    b := $b1
  end
end
```

The buffers between the actor are represented by only two actor state variables $\$b1$, $\$b2$. The actors A, B and C are rewritten as tuples where the elements are procedures, i.e. each procedure corresponds to an action.

Among the benefits from an actor based design is the fact that the components are completely decoupled, however the downside to this is that we need to find a scheduling and manage buffers for the communication. In the above example we show how those issues can be addressed for Cal actors.

3.2 Interface analysis

Modern type theory allows the programmer and the compiler to understand how different parts of a program fit or do not fit together. Statically, it can, for example, be determined if a method is called with the correct type of parameters or if the types of the l-value and the r-value in an assignment are compatible. Using type theory we can decide if the connections in an actor network are legal or not. However, this does only tell us if the communication patterns of the connected actors are compatible. When the network is actually executed, there may still be errors in terms of, for example, deadlocks. Therefore, it is desirable to have an interface theory that captures not only the static behavior, but also the dynamic behavior of an actor. The notion of behavioral types were introduced in [9], where the interface of an actor were described using interface automata [3]. It was shown how compatibility between and actor and a MoC can be statically determined.

Given a Cal actor with an internal schedule it is possible to extract an interface automaton which captures its dynamic behavior. Such an automaton can then be used for checking compatibility with other actors, but maybe more interesting, to check compatibility with a given MoC. In the example below the interface automaton for a simple actor is extracted.

EXAMPLE 13 (INTERFACE AUTOMATON). *The automaton that describes the interface of actor A3 below is shown in Figure 4. The labels are the corresponding action that is fired in the transition. For example, for the first transition to take place, the action s1 needs to fire, which in turn requires the presence of a token on the first input port, etc.*

```
actor A3[T] () T in1, T in2 T in3 ⟹ T out1, T out2 :
  s1 : action [a], [], [] ⟹ [c], [d] : ... end
  s2 : action [], [b], [] ⟹ [], [d] : ... end
  s3 : action [], [], [c] ⟹ [], [d] : ... end
  schedule
    s1(s2|s3)*;
  end
end
```

If we now wanted to use actor A3 in a given MoC, we could derive an interface automaton for that MoC and check compatibility by exploring the composite automaton.

This last section just touched upon the subject of interface automaton and showed how this can be extracted from a Cal

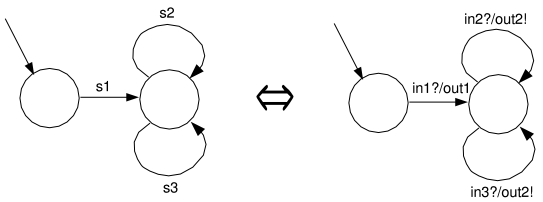


Figure 4: The data flow between actors can be analyzed by inspecting the connected output expressions and input patterns. Each of the transitions in the above automaton corresponds to the firing of an action. The corresponding interface automaton is shown to the right, where "a?" means waiting for a token on "a" and "a!" means writing a token to "a".

actor description. Ongoing work in area of component design in conjunction with interface automaton is found in [13], which also introduces Actif, which is a binary component model and a very suitable target for Cal code-generation.

3.3 Split phase execution

In the introductory example with the three versions of the Sum-actor, the last actor had a so called *split phase execution*. This means that the functionality of an actor is divided into two steps. First the actor calculates output values and then in the second step it updates its state variables. There are two main reasons for organizing the code this way:

- To minimize the latency from input to output by only doing the calculation necessary to get the output at the step and saving the rest of calculation for later. This is common practice in control systems were latency might be a key factor in achieving good performance.
- To handle MoC:s with iterations where an actor might be fired repetedly without changes in its internal state.

The task of the codegenerator is then to analyze the source code and determine the least amount of statements needed in order to compute the output. To do this it needs to calculate the dependencies for the output signals. In Cal this is fairly straightforward thanks to side-effect free expressions, limited mutability of data structures, and non-aliasing of variables.

4. LANGUAGE DESIGN: GOALS AND PRINCIPLES

Designing a programming language is an exercise in balancing a number of sometimes contradicting goals and requirements. The following were the ones that guided the design of Cal.

Ease of use Cal is intended to be a programming language, not an intermediate format or a representation for automatically generated code. Since we want people to actually write code in it, the notation must be reasonably convenient to write, with meaningful syntax rules, keywords, and structures. Because people make mistakes, it needs to be sufficiently redundant to allow effective error detection and localization, but simple and concise enough for frequent use, especially in frequently used areas. The language as such is a mix between imperative and functional programming languages. Functions and expressions are side effect free simplifying data flow analysis. However, since the most important goal in the design process has been to create a useful and expressive engineering tool, imperative statements, such as for- and while-loops have been included.

Minimal semantic core In spite of being a full-fledged programming language, we wanted to build Cal on a very small set of semantic concepts, for a number of reasons. First of all, being able to describe a large part of the full language through reductions to a smaller language makes the definition of language semantics much easier. From a practical perspective, this simplifies compiler construction—if there is a generic procedure that transforms any program into an equivalent program in the core language, then all it takes in order to compile the full language to any given platform is a code generator for the core language.

Focus and specificity Cal is a domain-specific language, that is aimed at providing a medium for defining actors. It was very important to draw a clear line between those pieces of functionality that were deemed to be part of an actor definition and those that were not. For example, in addition to clearly actor-specific structures such as actions and input/output patterns/expressions, expressions and statements were considered to be essential to defining an actor. On the other hand, there are many things that Cal explicitly does *not* contain, such as concepts for connecting actors, or mechanisms to aggregate actors into composites. The fact that Cal is explicitly agnostic about these issues makes it possible to use the language in a wide variety of contexts, which may provide very different designs in those areas.

Implementation independence Even though our first target for Cal actors is the Ptolemy II platform, we want the language to be retargetable, in the following two senses: First, we would like to be able to take an actor written, say, for Ptolemy II and be able to compile it to some other platform, say to some C code that runs against a different API. Secondly, we would like to enable other people to embed Cal into entirely different, but still actor-like, contexts, which have different kinds of objects (and types), different libraries, different primitive data objects and operators.

Making design knowledge explicit The key goal of Cal is to enable the author of an actor to express some of the information about the actor and its behavioral properties that are relevant to using the actor (e.g. verify its appropriate use inside a model, or to generate efficient code from it), but that would be only implicit in a description of the actor in a 'traditional' programming language such as C or Java.

5. CONCLUSION

This paper introduced a new actor language, Cal, for programming of embedded systems. The main characteristics of the language are the usage of port patterns, action guards, and action schedules to formally define legal execution paths. Using these it is possible to calculate the behavior of an actor in terms of the order that tokens are consumed and produced. This is then used in either the

scheduling actors or in proving compatibility between an actor and a MoC.

Block diagrams are a natural language for design and implementation of many embedded applications, e.g. the signal processing algorithms in mobile phones or the engine controller in your car. Such applications operate under strict timing requirement, while at the same time the computing resources are scarce. Determinacy and memory footprint are two important factors for any embedded application. Embedded system components in Cal allow a high degree of analysis to support the above requirements.

A Cal embedded system component is designed as a composite of interconnected actors which together with a MoC defines the functionality and the interface of a component. The information necessary to make such compositions is explicitly available for a network of Cal actors due to action and actor interfaces. The choice of MoC will determine the properties of the resulting composite. For example by choosing a more restrictive MoC, for example SDF, it is possible to perform offline analysis and calculate static schedules and memory consumption bounds.

There are currently two prototype Cal compilers. One that generates Java code for either the Ptolemy II framework [16] or the Grafchart environment [14]. A C-code generator is currently being developed at University of Lund, Sweden.

Acknowledgment

The authors wish to thank the members of the Ptolemy group, especially Edward Lee, Yang Zhao, and Chris Chang for fruitful discussions and joint work on several of the topics described in this paper. Also thanks to Ed Willink, Thales Research, for contributions to the compiler infrastructure and to Anders Blomdell, Lund University, for c-code generation input.

6. REFERENCES

[1] Benviniste A. and G. Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, volume 79. IEEE, September 1991.

[2] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, 1986.

[3] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component based design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software, First International Workshop, EMSOFT*, Lecture Notes in Computer Science, pages 148–165, Tahoe City, CA, USA, October 2001. Springer-Verlag Berlin Heidelberg.

[4] Johan Eker and Jörn Janneck. Cal actor language—language report (draft). Technical memorandum, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, 2002. http://www.gigascale.org/caltrop.

[5] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the ptolemy approach. *Proceedings of the IEEE*, 2002. To appear.

[6] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323–363, June 1977.

[7] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475, Paris, France, 1974. International Federation for Information Processing, North-Holland Publishing Company.

[8] Edward Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, pages 55–64, September 1987.

[9] Edward Lee and Yuhong Xiong. System-level types for component-based design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software, First International Workshop, EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 237–253, Tahoe City, CA, USA, October 2001. Springer-Verlag Berlin Heidelberg.

[10] Edward A. Lee. A denotational semantics for dataflow with firing. Technical Memorandum UCB/ERL M97/3, Electronics Research Lab, Department of Electrical Enginee and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, January 1997.

[11] Edward A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, 2002. to appear.

[12] Edward A. Lee et al. The Ptolemy Project. Department Electrical Engineering and Computer Sciences, University of California at Berkeley. http://ptolemy.eecs.berkeley.edu.

[13] H. John Reekie and Edward A. Lee. Lightweight component models for embedded systems. Technical Report UCB ERL M02/30, Electronics Research Laboratory, University of California at Berkeley, October 2002.

[14] Karl-Erik Årzén et al. Grafchart - a toolbox for supervisory level sequence control. Department of Automatic Control, Lund University, http://www.control.lth.se/grafchart.

[15] The Mathworks. *Simulink: Dynamic System Simulation for* MATLAB. The MathWorks Inc., Natick, MA, 2000.

[16] Lars Wernli. Design and implementation of a code generator for the cal actor language. Technical memorandum ucb/erl m02/05, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, 2002.