

# An introduction to the Caltrop actor language

**Johan Eker and Jörn W. Janneck**

Department of Electrical Engineering and Computer Sciences  
University of California at Berkeley  
Berkeley, CA 94720-1770, USA  
{johane,janneck}@eecs.berkeley.edu

## Abstract

Actors are computational entities that communicate with other actors and the environment by passing tokens via their input and output ports. Actors have state and parameters and when an actor is fired, input tokens are consumed, output tokens are produced, and the internal state is updated. Actors are connected to form models or applications. It is highly desirable to be able to investigate properties, such as bounded memory and absence of deadlock, for such applications. One key for doing this is to have full knowledge of the production and consumption patterns of the actors. Another issue for actor applications is to generate code for embedded systems. To do so is necessary to have a good understanding of control flow and memory management.

Caltrop is a novel actor language designed to facilitate the extraction of needed properties in order to handle those issues. It is a small domain specific language for implementation of actors. It is designed to be embedded in a richer environment, providing data types, operations and function libraries.

## 1 Introduction

This paper introduces the Caltrop actor language, created as a part of the Ptolemy II project at UC Berkeley [5]. The concept of actors was first introduced by Carl Hewitt in [4] as a means of modeling distributed knowledge-based algorithms. Actors have since then become widely used, for example see [1].

In Caltrop an actor is a computational entity with input ports, output ports, states and parameters. It communicates with other actors by sending and re-

ceiving *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.

Caltrop is a small domain-specific language for writing down the functionality of actors – including specifically their ports, their parameters, typing constraints, and firing rules. The goal 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 run-time 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 [6] process network model should be identical. The control of the run-time behavior such as scheduling and data transfer is left outside the language.

Caltrop is not intended as a full fledged programming language, but rather a small language to be embedded in an environment which provides necessary infrastructure. For example, we consider the definition of types to be outside the scope of the language and we instead provide a mechanism to import types and operations on them from an embedding environment. Similarly, even though functions and procedures are available in the language, the suggested usage model is to define functions and procedures in the host language in order to achieve good code reuse.

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 Caltrop approach is in part much inspired by the work presented in [8], where actors are formally defined using firing function and firing rules.

The type of a port may either be given explicitly or be parameterized using type variables. Constraints on the port types can be expressed and is then used to check type compatibility at run-time when the actor is connected to other actors.

Caltrop facilitates the use of several techniques for checking compatibility between connected actor. Production and consumption rates for a actor may be extracted from a Caltrop actor and can, for example, be used to statically check a synchronous data flow model [7] that is built using Caltrop actors. A more powerful way to analyze actor compatibility is to use automaton descriptions of the actor interfaces and analyze the behavior of composite system. One promising technique for doing this is interface automata [2], where actor compatibility is determined by matching possible legal input and output token sequences from the actors using their automata.

## 2 A small example

To get a flavor of the language, we will start with a quick example using a add-and-multiply actor. The idea is to design an actor that takes two numbers, adds them together, multiplies the sum with a constant  $k$  and then outputs the value. The actor will have two input ports, one output port and one parameter. It will operate on two, possibly infinite, input streams  $a$  and  $b$

$$\begin{aligned} A &= a_0, a_1, \dots, a_n \\ B &= b_0, b_1, \dots, b_n \end{aligned}$$

and produce a token stream  $C$ ,

$$C = c_0, c_1, \dots, c_n$$

which will have the following value

$$C = k(a_0 + b_0), k(a_1 + b_1), \dots, k(a_n + b_n)$$

When the actor fires it consumes tokens on input stream and produce a token on the output stream. There need to be at least one token available at both  $A$  and  $B$  in order for the actor to be fireable, and this is then the firing rule of the actor.

This actor implemented in Caltrop is shown below.

### Example 1

```

actor AddAndMultiply(Double k)
  Double A, Double B  $\implies$  Double C :
  action [a], [b]  $\implies$  [c] with Double c :
    c := k * (a + b);
  endaction
endactor

```

The signature of the AddAndMultiply actor declares one parameter  $k$ , two input ports  $A$  and  $B$ , and one output port  $C$ . The functionality of an actor is implemented using one or several *actions*, each associated with a *firing rule*. In this actor there is only one firing rule and only one action. The firing rule is given implicitly in the action signature as an *input pattern* '[a], [b]'. If the present tokens on the input ports match the action pattern, then the actor is fireable. In the case of the AddAndMultiply actor, it is fireable if there is one or more tokens on both input ports. When the actor fires, the first token at the ports  $A$  and  $B$  will be consumed and bound to  $a$  and  $b$ , respectively, and a token  $c$  will be produced and at the output port  $C$ .

In Example 1 the type of the actor is hardwired to type Double. Caltrop provides a way to create generic type actors by the use of *type variables*. Example 2 shows an alternative version of the AddAndMultiply actor. The type variable  $T$  is introduced and the inputs, outputs and parameters are declared to be of this type.

Another addition is that, the actor now has several firing rules and the actor may fire when there is one or more tokens at any of the input ports.

### Example 2

```

actor AddAndMultiply2[T] (T k) T A, T B  $\implies$  T C :
  action [a], [b]  $\implies$  [k * (a + b)] : endaction
  action [a], []  $\implies$  [k * a] : endaction
  action [], [b]  $\implies$  [k * b] : endaction
endactor

```

The pattern [a] will match if there is one or more token present and the pattern [] will match the any stream. This means that the firing rules above overlap and several possible actions may be fired at the same time. However, handling this ambiguity is outside the scope of the Caltrop semantics and must be decided by the environment. Caltrop allows the implementation of possibly nondeterministic actors and leaves it up to the code generation or run-time scheduler to resolve or disallow conflicting firing rules.

### 3 The actor definition

The header of an actor contains type parameters and actor parameters, its I/O signature and type constraints. This is followed by the body of the actor, containing a sequence of state declarations, definitions, actions, and initialization rules.

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.

By contrast, actor parameters are *values*, i.e. concrete objects of a certain type. They are bound to identifiers which are visible throughout the actor definition. Conceptually, these are immutable, i.e. they may not be assigned to inside an actor. A specific implementation such as the one in Ptolemy might change these parameters, but it has to ensure the consistency of the actor state with the new parameter values.

The I/O signature of an actor specifies the input and output ports, including their names, whether the port is a *multiport* or a *single port*, and the type of the tokens communicated via the port. While single ports represent exactly one sequence of input or output tokens, multiports are comprised of any number of those sequences (called *channels*). The names of the input ports are visible as variables inside the actor definition.

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 conditions are true. Type constraints may require types to be equal, or may require a type to be a subtype/supertype of another type. The following actor has type constraints on both its input and output types:

**Example 3 (Type constraints)** *The actor below has two inputs ports and one output port. The types of each of these ports are parameterized using the type variables  $T1$ ,  $T2$ ,  $T3$ . The type constraints express conditions on the relation of the allowed types.*

```
actor MyActor[T1, T2, T3] () T1 A, T2 B  $\implies$  T3 C
  where T1 < T2, T3 > T2 :
  procedure g(T1 arg1, T2 arg2) :
    ...
  end;
  a1 : action [a], [b, c]  $\implies$  [d] :
    ...
  end
  a2 : action [a, b], [c]  $\implies$  [d] :
    ...
  end
end
```

*The actions in this actor are labeled. The labeling is optional and used to refer explicitly to a particular action.*

#### 3.1 Actions

An action is an atomic piece of computation that an actor performs, usually in response to some input. The definition of an action needs to describe 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. If the actor signature is  $T A, T B \implies T C$ , as in Example 13, an input pattern may look like  $[a], [b]$  which binds  $a$  to the first token coming in on *input1* and  $b$  to the first one from *input2*. The same patterns may also be expressed using the port names:  $input1 :: [a] input2 :: [b]$ . 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.

#### 3.2 Action matching

An actor can consist of any number of action definitions. 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. An actor can

only select an action that *matches* the current input in the current state. Such an action is said to be *irable*. The parts of an action definition that are considered during matching are the following:

- The input patterns.
- The evaluation of the *guard*, described in the where-clause.

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 action matches an input in a given state if and only if

- The sequential reading of tokens as constrained by the variable dependencies finds enough tokens on each selected channel to bind the token variables.
- The guard, i.e. the expressions in the where-clause all evaluate to true.

In case more than one action is irable at some point, disambiguation is left to the environment.

**3.2.1 Input patterns:** 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),
- whether these 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 this  $[a, b, c]$ . 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.

Sometimes, it is necessary to read a few tokens from the input, and additionally be able to query the rest of the input sequence. This can be achieved by the

pattern  $[a, b, c|s]$ , which This pattern binds  $a, b,$  and  $c$  again to the first three input tokens, and  $s$  to the sequence representing the rest of the input sequence. The type of  $s$  is  $\text{Seq}[T]$ , where  $T$  is the token type of the port.

**Example 4 (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  $[a, b, c, d | e]$  also matches, binding  $a, b, c,$  and  $d$  to 1, 2, 3, and 4, respectively, and  $e$  to the empty list  $[]$ .

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

**3.2.2 Multiport patterns:** A multi input port is a mapping from channel identifiers, of type `ChannelID` to sequences of tokens. A common solution is to use integers as channel identifiers, i.e. the channels are identified by a number and in order to access a channel the multiport variable is indexed by the channel number.

Caltrop supports the all the usual facilities for maps which are need for dealing with multiports. For example, `dom p` computes the set of channel identifiers defined for the port  $p$ , i.e. the domain of the map defining the multi port. The variable  $p[a]$  is the channel of port  $p$  identified by  $a$ .

Multiports have any number (including zero) of different *channels*, i.e. individual sub-ports which are independently associated with their own input sequences. Actors may want to read from all of these sequences, any of them, or from specific ones. Multiport patterns are similar to single port patterns, except that they contain a specification of which channels the pattern is to be applied to. This *channel selector expression* is either an expression evaluating to a collection of channel identifiers or one of two keywords `all` and `any`. The syntax is a port pattern followed by the selector expression enclosed in curly braces, .e.g.  $[a,b]\{1,3\}$  which matches the two first tokens on channel 1 and 3.

**Example 5 (Multi port pattern)** The *Add* actor below has one multi input port and when fired it sums up the value of the input tokens and produces a output token. The first action is irable if there are tokens present at channel 1,2, and 3. The second action may fire if there all tokens on all connected channels. Finally the third action may fire if there is any token available at all. The type of the channel identifiers in this example is assumed to be integers.

Please note that the input conditions for the actions overlap and it is again left to the environment to handle this ambiguity.

```

actor Add () multi Double input  $\implies$  Double output :
  a1 : action [a]{1,2,3}  $\implies$  sum :
    sum := a[1] + a[2] + a[3]
  end
  a2 : action [a] all  $\implies$  sum with Double sum := 0 :
    foreach Integer i in dom a :
      sum := sum + a[i]
    end
  end
  a3 : action [a] any  $\implies$  sum :
    foreach Integer i in dom a :
      sum := sum + a[i]
    end
  end
end

```

**Example 6 (Multiport pattern)** Consider the following actor that select one token from one of its input ports and outputs that token.

```

actor Selector[T]() multi T input, ChannelID select
   $\implies$  T output :
  action [a]{s}, [s]  $\implies$  [a[s]] : end
end

```

The pattern match if

- the select port has a token and
- the input channel determined by that token has a token.

The effect is to bind the variable sel to the channel selection token, and a to the token read from the multiport.

**3.2.3 Repeat patterns:** 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 7 (Repeat Pattern)** This actor upsamples an input stream by an integer factor by inserting tokens with value zero. The upsample factor is given by the factor parameter. On each firing, this actor reads one token from the input 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  $\implies$  T output :
  action [a]  $\implies$  [b] repeat factor :
    b := [if i = phase then a else 0 end :
      for i in [1..factor]
    endaction
  endactor

```

**3.2.4 Action guards:** The input conditions for an action must not only be defined by the pattern but also using a guard expression, e.g. **action** [a],[b]  $\implies$  ... **where** a > b

This action may only fire if there the variables a and b may be bound to tokens and if the value of a is greater than the value of b.

The where-clause contains a set of Boolean expressions that may impose additional conditions on the values of the variables bound by this process. For example, the following constraint states that the value of the token on the second input port must be greater than the value of token on the first input port:

**Example 8 (Guards)** This sort actor has two action that sorts the input tokens according to a given sorting criterion, i.e. the function f which is a parameter. If the value of the evaluation of f is true the first action is chosen otherwise the second.

```

actor Sorter[T]([T  $\longrightarrow$  boolean ] f)
  T input  $\implies$  T output1, T output2 :
  action [a]  $\implies$  [b], [] where f(a) : end
  action [a]  $\implies$  [], [b] where not f(a) : end
endactor

```

### 3.3 Channel and port expressions

The output of actors is described by expressions defining lists of tokens for each output channel. However, here the situation is a little simpler, as the output is simply described by ordinary expressions.

The expressions computing the output are ordinary expressions, except for one detail: They can refer to two version of each state variables, the one at the beginning and the one at the end of the action. The state at the beginning of an action is referred to be its usual name, while the state at the end of the action is referred to by adding the tick (') to the variable name.

**Example 9 (State variable)** This simple actor calculates the current and the previous token. The result is the difference between the new state and the old state being output as a token.

```

actor Diff [T]() T input  $\implies$  T output :
  T s;
  action [a]  $\implies$  [s' - s] :
    s := a;
  end
endactor

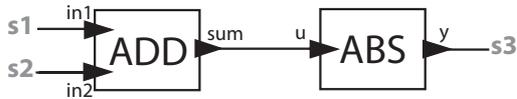
```

## 4 Composition

Caltrop says nothing about how actor are composed together. The semantics of a network of interconnected Caltrop actors is almost entirely left to the framework in which the actors are instantiated. The meaning of the Caltrop model in Figure 1 does not only depend on the actors A, B and C but also on the order in which they are executed and the way their communication is handled.

In this section we will explain some of the benefits received when using Caltrop. Many of those are more or less open research issues, such as determining the behavior of composed actors.

We will start by explaining about the use of the type conditions and then continue with some ideas about analysis of composite behavior.



**Figure 1:** This block diagram defines a Caltrop application consisting of two actors Add, which adds the values of the input tokens and outputs the sum, and Abs, which calculates the absolute value of its input.

### 4.1 Type Checking

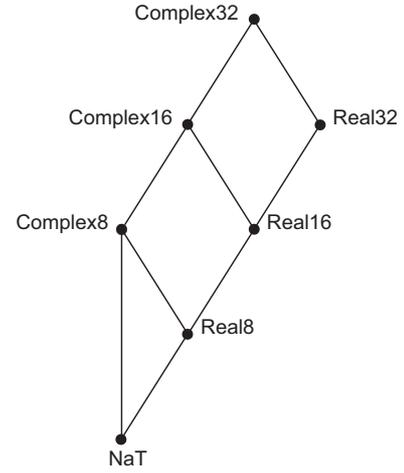
The type system in Caltrop is, as mentioned previously, regarded as part of the execution environment. The environment provides a set of types, operations on variables of these types and a type lattice, which defines the relations between different types. Figure 2 shows an example of a type lattice.

Consider the following implementation of the actors in Figure 1. We will below give some examples on how the type constraints in the below actors can be used to infer types or check types.

```

actor ADD[T1, T2, T3] () T1 in1, T1 in2  $\implies$  T3 out
  where T1 = T2, T1  $\leq$  T3 :
  action [a], [b]  $\implies$  [a + b] :end
end

```



**Figure 2:** A type lattice expresses a partial order between types in a type system. The lines represent the relation  $\leq$  and the higher up in the lattice the greater the type is. In this lattice  $Real16 \leq Real32$ .

```

actor ABS[T1, T2] () T1 u  $\implies$  T2 y
  where Complex8  $\leq$  T1, T2  $\leq$  Real32 :
  action [a]  $\implies$  [b] :
    b := Sqrt(Im(a) * Im(a) + Re(a) * Re(a));
  end
end

```

**Example 10 (Type inference)** Type constraints can be used both for inferring type in the case where the type variables are unbound or they can be used to check the validity of type values. In Ptolemy signals may automatically converted if there exist a way to do this without losing information. For example, a 8 bit real value may be converted into a 16 bit value without any loss. In general a type  $T$  may be converted into  $T'$  without any loss only if  $T \leq T'$ .

From the system in Figure 1 and the type constraints in the actors the following system of equations is derived.

$$ADD: \begin{cases} T_{in1} = T_{in2} \\ T_{in1} \leq T_{out} \end{cases} \quad ABS: \begin{cases} T_u \geq Complex8 \\ T_y \leq Real32 \\ T_y \leq T_u \end{cases}$$

Now assume that the types of the signals  $s1$ ,  $s2$  and  $s3$  are the following

$$Connections: \begin{cases} T_{s1} \leq T_{in1} \\ T_{s2} \leq T_{in2} \\ T_y \leq T_{s3} \\ T_{out} \leq T_u \end{cases} \quad External: \begin{cases} T_{s1} = Complex16 \\ T_{s2} = Complex8 \\ T_{s3} = Real32 \end{cases}$$

Solving the above equation for the lowest upper bound on the types yields the solution  $T_{in1} = T_{in2} = T_{out} = T_u = Complex16$  and  $T_y = Real16$ .

**Example 11 (Type checking)** *The use of type inference is very convenient. Actors may be composed without the programmer having to care for all the details. However, the downside is that when there are errors they tend to propagate through the model and are often reported far away from their source, making debugging cumbersome. Also in many applications, for embedded systems, the data types are of great concern and cannot be left to be determined automatically. In Caltrop the type constraints can also be used to check the validity of type parameters that are explicitly given when the actors instantiated.*

- `new Add[Real32, Real16, Real8]();`  
*Wrong, the type conditions are violated*
- `new Add[Real16, Real16, Real32]();`  
*Correct, the type conditions are fulfilled*

**4.1.1 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 by inspection of the input patterns and output expressions of the connected actors. Consider the model given in Figure 3, where the Caltrop code for the actors is found below.

```
actor A1[] () Double in1, Integer in2
  => Double out1, Integer out2 :
  action [a], [] => [], [d] : ... end
  action [], [b] => [], [d] : ... end
end
```

```
actor A2[] () Double in3, Integer in4
  => Double out3 :
  action [a], [b] => [c] where a > b : ... end
  action [a], [b] => [c] where b ≤ b : ... end
end
```

In this trivial case is straightforward to see that A2 will never produce any output, because there will never be an output from A1 that will match any of the actions of A2. The model in Figure 3 will hence deadlock.



**Figure 3:** The data flow between actors can be analyzed by inspecting the connected output expressions and input patterns.

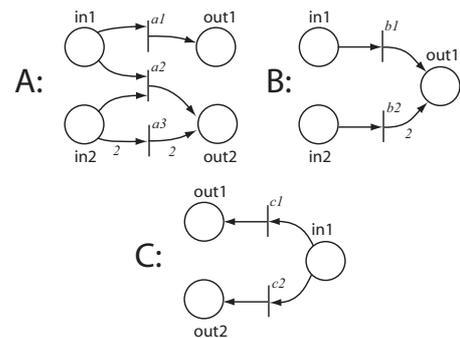
The example above demonstrated how the action signatures can be used to determine possible production and consumption rates for an actor. The gen-

eral 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 Caltrop in this setting is to make it straightforward for analysis tools to extract the needed information from the Caltrop actor.

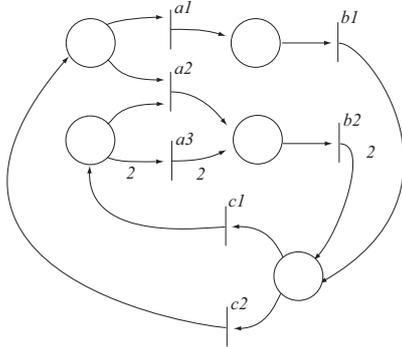
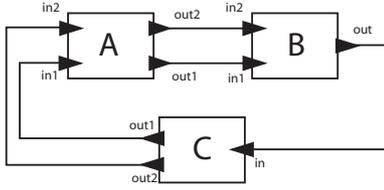
**Example 12** *To demonstrate the ideas further we will here give a more extensive example. Consider the following three Caltrop actors:*

```
actor A[T] () T in1, T in2 => T out1, T out2 :
  a1 :action [a], [] => [c], [] : ... end
  a2 :action [a], [b] => [], [d] : ... end
  a3 :action [], [b, c] => [], [d, e] : ... end
end
actor B[] () T in1, T in2 => T out1 :
  b1 :action [a], [] => [c] : ... end
  b2 :action [], [b] => [c, d] : ... end
end
actor C[] () T in1, T in2 => T out1 :
  c1 :action [a] => [a], [] where a > 0 : ... end
  c2 :action [a] => [], [a] where b ≤ 0 : ... end
end
```

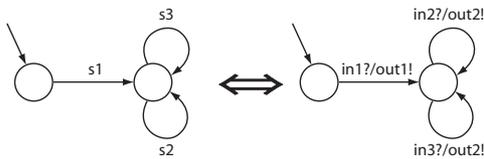
*The production and consumption of tokens can be expressed using the following petri net models, where places corresponds to inputs and output ports and transitions correspond to actions.*



*When the above actors are composed to a model (below) the resulting petri net can be composed using the petri nets for the actors. This idea is now to investigate the petri net for the composite for properties such as deadlock and memory bounds.*



**4.1.2 Behavioural analysis:** It is possible to constraint the order in which actors are allowed to be fired. The selector clause in Caltrop provides a way to constraint the action sequence through giving a regular expressions over the action labels. In the actor A3 below the order the three actions s1, s2, and s3 may be fired is constraint by the selector. The resulting automaton 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.



**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.

```

actor A3[T] () T in1, T in2 T in3  $\implies$  T out1, T out2 :
  s1 :action [a], [], []  $\implies$  [c], [d] : ... end
  s2 :action [], [b], []  $\implies$  [], [d] : ... end
  s3 :action [], [], [c]  $\implies$  [], [d] : ... end
  selector
    s1(s2|s3)*;
  end
end

```

## 5 Code generation

It is common for the generated code us have a so called split phase execution. This means that the firing of the actor is divided into two steps. In the first step the actor calculates output values and in the second it will update its state variables. There are several reasons for organizing the code this way:

- Minimize latency
- Iteration
  - When the actor is fired several times in fire must not change the state of the actor.

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 Caltrop this is fairly straightforward thanks to the side-effect free expressions, i.e. a variable may only change value as the result of an executed statement.

## 6 Expression and Statement Description

This section gives an informal overview of Caltrop. We will go through the different parts of the language and demonstrate the usage by a number of examples. For a more thorough and formal description of Caltrop see [3].

### 6.1 Data types

As mentioned previously Caltrop is designed to be embedded in a host environment or host language. Most types in Caltrop, including Integer, Double, and String must be imported from this host environment. The choice of these types, as well as the operations on them is left to the same environment. The data types (*Double*, *Integer*) and the operations (**mod**, **+**, **-**) in Example 13 are all not part of Caltrop.

#### Example 13 (Types)

```

actor a1 () Double d1, Double d2  $\implies$  Double d3 :
  Integer n := 0;
  function f(Double d1, Double d2, Integer i)  $\longrightarrow$  Double :
    if i mod 2 = 0 then d1 else d2 end;
  end;
  action [a],[b]  $\implies$  [c] with Double d :
    c := f(a, b, i) - 1;
    n := n + 1;
  end
end

```

Caltrop has tuples to express composite data structures. A tuple is an expression which consists of a list of variables enclosed by brackets. Example 14 shows a function which returns a tuple consisting of an integer, a boolean and a string.

#### Example 14 (Tuples)

```
function myFun() → [Integer, boolean, string] with
  Integer a = 3, boolean b := true, string s := "ptolemy" :
  (a, b, s)
end
```

Caltrop does define a small number of built-in parametric aggregate types such as Set, Map, and List. A map  $m$  from integer (the key) to double (the value), list  $l$  of doubles and a set  $s$  of integers is declared and assigned below.

```
map[Integer, Double] m;
list[Integer] l;
set[Integer] s;
m := {1 → 4, 2 → 7};
l := [1, 2, 3];
s := {1, 2, 3};
```

Caltrop use comprehensions to iteratively construct sets, lists, or maps. The syntax and semantics is very similar in all three cases, see Example 15 and Example 16.

A comprehension consists of an expression followed by one or several *generators*, which are followed by zero or more *filters*. A list comprehension has the following form [ *expressions* : *generators*, *filters* ];

In Example 15 The generators introduce new variables, and successively instantiate them with the elements of the collection after the *in* keyword. The expression computing that collection may refer to the generator variables defined to the left of the generator it belongs to.

Once all generators have produced a variable assignment, the element expressions (those appearing to the left of the colon) are evaluated and the resulting values are added. After that, a new assignment is computed by first taking a new value for the rightmost generator variable, until that collection is exhausted, whereupon the next generator on the left is advanced etc., until the leftmost generator is exhausted and the process terminates.

**Example 15 (Set comprehensions)** *The expression {1, 2, 3} denotes the set of the first three natural numbers, while the set {2 \* a: for a in {1, 2, 3}} contains the values 2, 4, and 6. Finally, the set {a \* b: for a in {1, 2, 3}, for b in {4, 5, 6}, b > 2 \* a} contains the elements 4, 5, 6, 10, and 12.*

List comprehensions have the same syntax and work in a similar fashion, except that the order is of course relevant to lists. Element expressions are added in left-to-right order to the list at each iteration.

Map comprehensions again work similarly, but they construct map objects, i.e. finite mappings from *keys* of one type to *values* of another. Instead of element expressions as in the previous two comprehensions, we have *mappings*, describing the key/value pairs to be added to the map. The use of map comprehensions is shown below.

**Example 16 (Map comprehensions)** *The following map comprehension*

```
map{a → 2 * a: for a in {1, 2, 3}}
creates the map {1 → 2, 2 → 4, 3 → 6}.
```

## 6.2 Expressions

Expressions in Caltrop are side-effect-free and strictly typed. The following gives an overview of the kinds of expressions and expression syntaxes provided by Caltrop. Literals are constants of various types in the language, for example boolean values such as true or false, numerical values such as integers, 2 or reals, 3.14, or strings, "a string".

The *if*-expression used shown in Example 13 has the following syntax:

```
a := if Expression then Expression else Expression end;
```

The first subexpression must be of type boolean, and the value of the entire expression is the value of the second subterm if the first evaluated to true, and the value of the third subterm otherwise.

A *let*-expression provides a way to introduce local names for the values of expressions. This is often useful to factor out large subexpressions that occur several times.

```
let Double a = 3.5, T b :
  expression
end
```

The type of the *let*-expression is the type of the expression that constitutes its body.

In Caltrop functions and procedures are defined using function or procedural closures. A function closure is the result of the evaluation of a *lambda*-expression. It represents a function that is defined by some expression, and it's parameterized and may refer to variables defined in the surrounding context. Functions are side-effect free and may thus not contain any statements. In general, however, they may refer to stateful variables and thus may themselves depend on the assignment of variables in their con-

text.

**Example 17 (Function definition)** A function variable  $f$  is declared and assigned the value of a function closure:

```
[Double, Integer → Double] f;  
f := lambda (Double d, Integer n) → Double  
  with list[Double] l = [d : for Integer i in Integers(1, n)] :  
  Mul(l)  
end
```

A more straightforward shorthand notation is

```
function Mul(list[Double] l) → Double :  
  if #l = 1 then l[1] else l[1] * Mul(Tail(l)) end  
end
```

The function  $f$  is applied as:

```
Double d := f(3.14, 1);
```

If the types of the formal parameters are  $T_1$  to  $T_n$ , respectively, and the return type is  $T$ , then the type of the function closure is  $[T_1, \dots, T_n \rightarrow T]$ .

Functions come in two forms: they are either the result of evaluating a lambda-expression, or are provided as part of the environment. There is no difference in the way they are used inside expressions, but of course they differ in the way they are evaluated, and also in the way their types are determined. A function application simply is an expression of the form  $E(E_1, \dots, E_n)$ , where  $E$  must evaluate to a function taking  $n$  arguments of the correct type.

Procedural closures are somewhat similar to function closures, in that they encapsulate a piece of code together with the context in which it was defined. However, in the case of block closures, this piece of code is a list of statements, i.e. executing a block closure is likely to have side effects (as opposed to the application of a function closure). Example 3 shows an example of a procedure definition. The type of the procedure  $g$  is  $[T_1, T_2 \rightarrow]$ . Since block closures can produce side effects, their execution cannot be part of the evaluation of an expression.

An indexer is a side-effect-free way of extracting a value from a specified *location* inside another object. For instance, the built-in types `List[T]` and `Map[K, V]` support indexing functions: a list can be indexed by an integer, and produces the corresponding element in that list as a value. Similarly, a map can be indexed by a key, and produces the corresponding value for that key.

**Example 18 (Indexing)** A map  $m$  from Integer to Double is indexed as Double  $d := m[2]$ .

In a similar fashion, the environment may provide other kinds of data objects that support indexing.

### 6.3 State variables

State variables are declared inside the actor body, see Example 13. The expressions used to define the initial value of a state variable can refer to parameters, ports, and state variables.

State variables come in two forms: they are either *assignable*, or they are *mutable*. Assignable state variables may be directly assigned to in assignment statements, i.e. they can occur by themselves on the left-hand side of an assignment, e.g.

```
v := a + b
```

By contrast, mutable variables may never occur by themselves on the left-hand side of an assignment. The next section discusses mutable variables, and a related notion, mutable objects.

Conceptually, a mutable variable represents a number of assignable *locations*. In order to assign to these locations, and also in order to access them, the variable must be *indexed*.

In Caltrop, an index into a variable is an object (often a tuple) enclosed in square brackets. An assignment into a mutable variable thus may look like this:

```
v[k, x * y] := a
```

A state variable that is declared to be mutable must be of a type that supports mutation. In order to support mutation, a type must have an (*indexed*) *mutator*, i.e. a procedure that changes specific locations in the object, which is invoked when such a location is assigned to. A type that supports mutation is called a *mutable type*.

### 6.4 Statements

The execution of an action as well as initialization, happens as a sequence of statements, each of which may change the state of the actor. Several statement constructs (and in fact action definitions themselves) may contain with-clauses, which serve to define new variable symbols that are local to the scope in which the with-clause occurs.

A with-clause may also contain *definitions* which are constant in the scope, i.e. they cannot be assigned to and neither are they mutable.

Assigning new a new value to a variable is the fundamental form of changing the state of an actor.

Each identifier must be an assignable variable in the current context, and each indexed identifier must be a mutable variable, and of course its mutator must support the index type. Each non-indexed identifier

may occur only once. If an indexed identifier occurs more than once, the order in which the assignments are executed is not specified.

Caltrop provides a number of control flow constructs to write iterative programs and control the flow from statement to statement. We distinguish between *branching* constructs, which choose between a number of possible statement sequences based on some condition, and *iteration* constructs which repeatedly execute a set of statements.

**6.4.1 Guarded assignment:** The guarded assignment statement is used to branch based on the actual data types of a number of objects. This is a way of doing well defined type castings. The syntax is as follows:

**Example 19 (Guarded assignment)**

```
assign exp1, exp2 to
  T1 a, T2 b : stmts : end
  T2 a, _ : stmts : end
default : stmts : end
end
```

Following the expression list is a list of cases, which in turn consist of a list of declaration patterns (the number of patterns must be the same as the number of expressions) followed by a sequence of statements. The statements are executed if the patterns *match* the values computed by the expressions, where values and patterns are matched according to their position.

A declaration pattern is either the underline character or a variable declaration consisting of a type and a variable symbol. An underline character matches any object, and does not introduce a new variable. A declaration matches only objects of the given type (or any subtypes of it). If it matches, it binds the corresponding object to the variable symbol, which will be visible inside the sequence of statements belonging to this case.

**6.4.2 While:** The while-construct repeats execution of the statements as long as a condition specified by a Boolean expression is true.

**Example 20 (While)**

```
Integer i := 0;
while i < 10 with Integer j := 0 :
  j := j + 1;
  i := j;
end
```

**6.4.3 Foreach:** The foreach-construct allows to iterate over a collections, successively binding vari-

ables to the elements of the expression and executes a sequence of statements for each such binding.

**Example 21 (Foreach)** A for loop that iterates over two lists:

```
List[Integer] l1 := {2 * a : for a in [0..5]};
List[Integer] l2 := {1, 2};
Integer sum := 0;
foreach e1 in l1, e2 in l2 :
  sum := sum + e1 * e2;
end;
```

This loop will calculate the sum:  $0 * (1 + 2) + 2 * (1 + 2) + 4 * (1 + 2) + 6 * (1 + 2) + 8 * (1 + 2) + 10 * (1 + 2)$ . The expression `[0..5]` creates a list of integers in the specified range.

Variables are bound from left to right, and the expressions defining the collections the variables are iterated over may use the values bound to the variables to their left.

## 7 Summary

Caltrop is a small domain specific language designed to support the implementation of actors and the analysis of networks of such actors. 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 of the language has been to create a useful and expressive engineering tools imperative statements, such as for- and while-loops have been included.

## References

- [1] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, 1986.
- [2] 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. Springer, October 2001.
- [3] Johan Eker and Jörn Janneck. Caltrop reference manual. Technical Memorandum UCB/ERL ???/??, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, November 2001.

- [4] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–363, June 1977.
- [5] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong. Heterogeneous concurrent modeling and design in java. Technical Memorandum UCB/ERL M01/12, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, USA, March 2001.
- [6] 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.
- [7] E. Lee and D. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, pages 55–64, September 1987.
- [8] Edward A. Lee. A denotational semantics for dataflow with firing. Technical Memorandum UCB/ERL M97/3, Electronics Research Lab, Department of Electrical Engineer and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, January 1997.