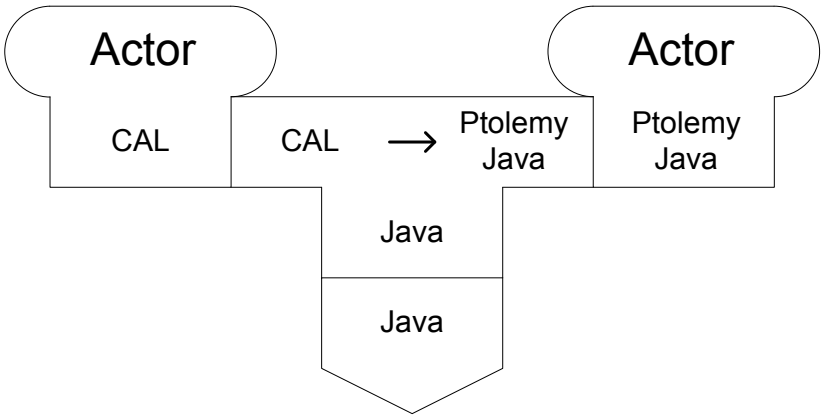
Design and implementation of a code generator for the CAL actor language

DIPLOMA THESIS

Lars Wernli



Supervisor:
Dr. Jörn W. Janneck
EECS Department
University of California at Berkeley

Professor:
Prof. Dr. Lothar Thiele
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Zurich

Contents

Chapter 1

Introduction

Ptolemy II is a software system written in Java for modeling and simulation of embedded real-time systems. Ptolemy's focus is on supporting a rich variety of models of computation, which deal with concurrency and time in different ways. Models in Ptolemy II are *hierarchical* and *heterogeneous*. They can consist of several subsystems which can be based on different models of computation as shown in figure 1.1.

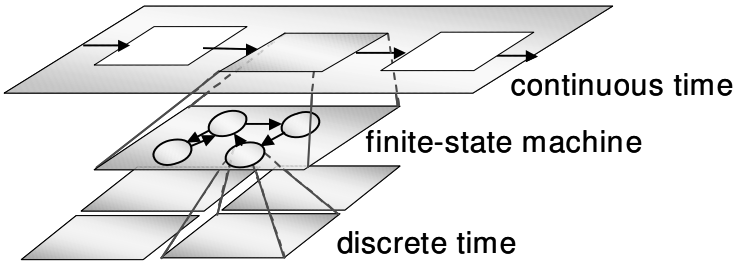


Figure 1.1: Hierarchical and heterogeneous modeling in Ptolemy II.

Ptolemy II takes a *component based* view on design. Models are constructed from components which we will refer to as *actors*. In the context of this work, an actor is a computational entity, which consumes sequence of *tokens* at its *input ports* and produces sequences of tokens at its *output ports* as a function of the actors current *state* and *parameters* and the incoming token sequences. It also changes its state, or rather computes a successor state.

Figure 1.2 shows a simple actor before and after a firing. The `CharReader` actor has two input ports (N and Data). When fired, it is supposed to read one token from port N. The value of this token specifies how many tokens to read from the `Data` port. In this case, the value read from `N` is 3 and the actor reads the three character tokens `'C'`,

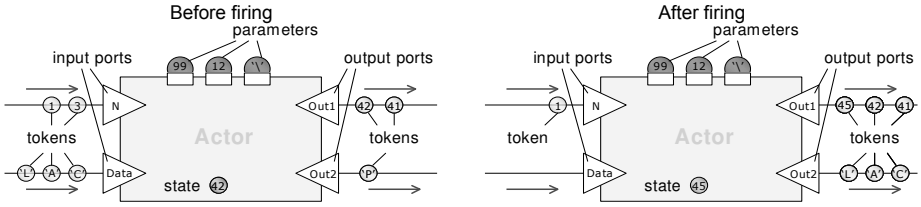


Figure 1.2: The *CharReader* actor before and after *firing*

`'A'` and `'L'` from `Data`. The actor has a state variable `state`, which at each firing is incremented by the value of the token read form `N`. In this firing it is changed from 42 to 45. The actor has two output ports `Out1` and `Out2`. At each firing it outputs the current value of `sum` at `Out1` and outputs the tokens read from `Data` at `Out2`.

Atomic actors in Ptolemy II have so far been written in Java. Writing actors in Java requires a certain knowledge about the Ptolemy II API which poses a considerable entrance barrier for new authors. Actor writing can be error-prone and repetitive even for experienced authors, and it forces the author to take certain design decisions on issues which arise only due to the Ptolemy API, but do not really concern the semantics of the actor. Chapter 2 will focus on those issues in detail, after giving a general overview over the Ptolemy II software.

An alternative solution is to write actors in *CAL* - a domain specific language for writing data flow actors. CAL is not a general purpose language, but is supposed to be embedded into a richer environment. It is supposed to be platform independent and retargetable to a rich variety of target platforms. It provides a strict semantics for defining an actors computational operations, ports and parameters and it provides a set of composite data structures. But it leaves certain issues to the embedding environment, such as the choice of supported data types and the definition of their semantics. An introduction to the CAL language will be given in chapter 3.

This work addresses the design and implementation of a code generator which allows generating atomic actors in Ptolemy II from an actor specification in CAL. Chapter 4 will give the big picture of the CAL compiler, as it is shown in figure 1.3, and it will explain how the structure of the compiler makes it easy to re-target it to other platforms.

In order to make the compiler easy to re-target and the code generation as easy as possible to implement, the compiler transforms the CAL specification of the actor into a subset of the language, which still fully represents the semantics of the actor. Chapter 5 gives an idea about the transformations performed and explains some of them more in detail.

What does a generated actor look like? And how does it have to be structured in order to be reusable for other platforms? Chapter 6 will focus on the design of the target code in detail, and explain how re-usability is achieved by separating the generated code into

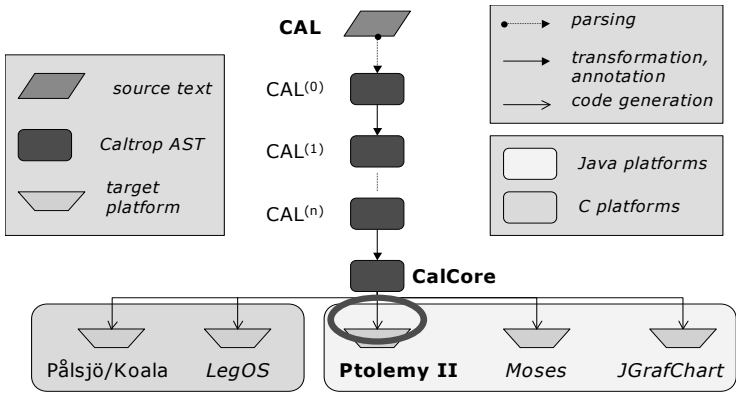


Figure 1.3: The big picture of the CAL compiler

a generic and a platform specific part.

Chapter 7 focuses on specific implementation details of the code generator itself. It explains how the code generator traverses the source program and how it uses an intermediate representation of the source which allows writing the generated code in a non sequential way.

In Chapter 8 we finally summarize the achievements of this work and explain opportunities for further work related to the CAL code generation.

Chapter 2

Ptolemy

Ptolemy II is a software system written in Java for modeling and simulation of heterogeneous systems. This chapter briefly introduces some aspects of Ptolemy which are important to understand the context of this report, and it motivates the actor description language *CAL*, which will be introduced in the next chapter. A detailed documentation of the Ptolemy project can be found in the most recent overview paper [5].

2.1 The Ptolemy Project

The *Ptolemy Project* at UC Berkeley studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on embedded systems, particularly those that mix technologies including for example analog and digital electronics, hardware and software, and electronics and mechanical devices. The focus is also on systems that are complex in the sense that they mix widely different operations, such as signal processing, feedback control, sequential decision making, and user interfaces.

During the last years, the Ptolemy group has constructed a simulation and modeling software system in Java. The current version of this software is called *Ptolemy II* and it enables simulation of heterogeneous systems. Ptolemy II includes a graphical user interface called *Vergil* for visual depiction and construction of models. Figure 2.1 shows an example of a Ptolemy II model displayed as a block diagram in Vergil. The example is a *discrete time* model of a simple communication system, and its functionality is briefly commented in the model display. Figure 2.2 shows the execution of the model in the *run window*.

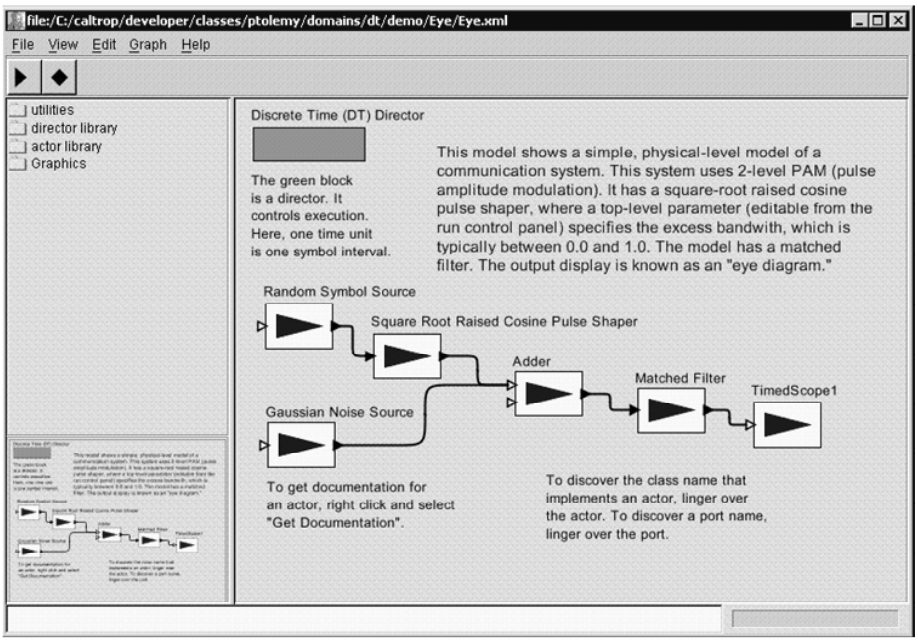


Figure 2.1: A Ptolemy II model displayed in the graphical user interface Vergil. This *discrete time* model consists of six *actors* and the discrete time *director* which schedules the actor executions.

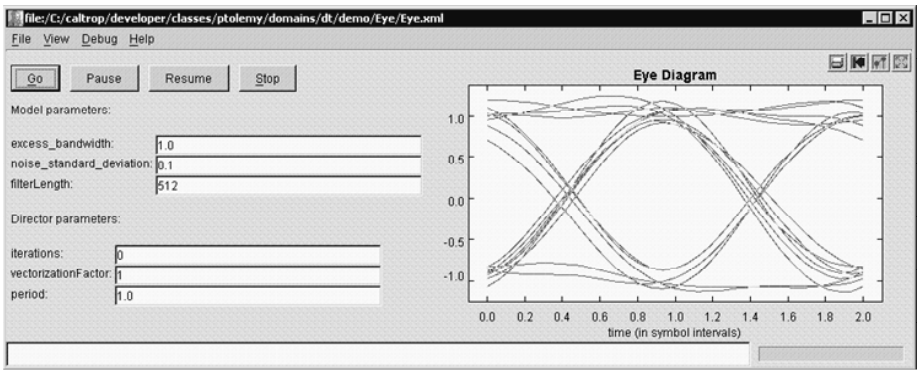


Figure 2.2: The model shown in figure 2.1 is executed in the *run window*. The run window allows the user to control execution of the model and to set the model parameters. It displays the graphical output produced by the `TimedScope1` actor, which is the eye-diagram known from analog signal processing.

2.2 Hierarchical and Heterogeneous Modeling

Ptolemy supports a variety of computational models, such as *continuous time*, *discrete time*, *discrete events*, *synchronous data flow* and many others. We will refer to the implementation of a computational model as a *domain*. Each domain has a director which controls the interactions between the actors and schedules the actors for firing. The simulation of the *discrete time* model shown in figure 2.1 is controlled by the *discrete time director*.

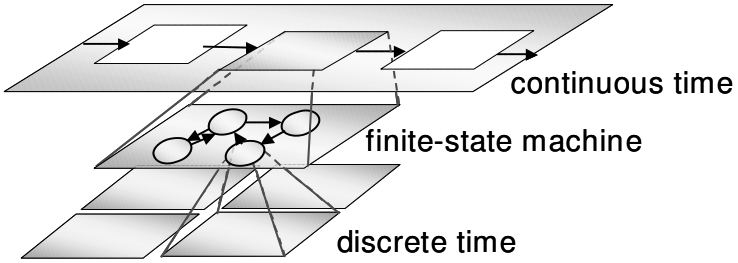


Figure 2.3: Hierarchical and heterogeneous modeling in Ptolemy II.

As shown in figure 2.3, a Ptolemy II model can consist of several sub-models which can have different domains. Mixing different domains in one and the same model is what we call *hybrid modeling*. The capability to simulate such hybrid models is the key feature of Ptolemy II and its main 'selling point' in comparison to other existing modeling software systems.

2.3 Actor based modeling

Ptolemy II takes a component view of design, in which models are constructed as a set of interacting components which we refer to as *actors*.

The concept of actors was first introduced by Carl Hewitt in [8] as a mean of modeling distributed knowledge-based algorithms. Actors has since then become widely used, for example see [2]. The CAL approach of defining the actor concept is in part much inspired by the work presented in [11].

In the context of CAL, an *actor* is a computational entity with input ports, output ports, state and parameters. It communicates with other actors by sending and receiving tokens (atomic pieces of data) through its ports. A CAL actor contains one ore more *actions*. An action defines a computation, which consumes sequences of tokens from the actor's input ports, and produces sequences of tokens at its output ports. The execution of an action may change the actor's internal state, and the produced output sequences are functions of the current actor state as well as of the consumed input sequences. An

action may contain a set of *guard conditions*, which are expressions of type boolean and pose necessary conditions for the action to be executed. The execution of one of the actor's actions is called *firing*.

A Ptolemy II actor can be specified in two different ways: Either by composing several other actors into a sub-model - these actors are called *composite* actors - or the actor can be *atomic*. Atomic actors are (so far) coded in Java, the implementation language of Ptolemy II. Of course, every composite actor's functionality could just as well be defined by an atomic actor.

This report exclusively deals with atomic actors, and from now on if we use the word *actor* in the context of Ptolemy, we mean *atomic actor*.

2.4 Writing atomic actors in Ptolemy II

This section gives a short introduction to actor design in Ptolemy II and discusses some drawbacks resulting from specifying actors in Java. A more detailed documentation of actor design is given in [5].

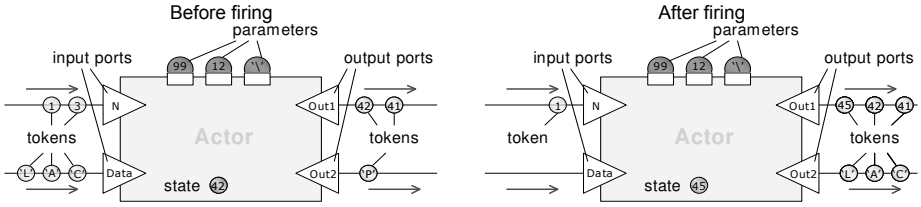


Figure 2.4: An *actor* before and after *firing*

Figure 2.4 shows the `CharRead` actor which was already introduced in section 1. Remember that the actor has two input ports (N and Data). When fired, it is supposed to read one token from port N, and this token specifies how many character tokens to read from the `Data` port. The actor has a state variable `state`, which at each firing is incremented by the value of the token read from `N`. The actor has two output ports `Out1` and `Out2`. At each firing it outputs a token with the current value of `state` at `Out1` and outputs the tokens read from `Data` at `Out2`. The actor has three parameters `max_N`, `max_state` and `endChar`. Assume the actor cannot fire if the value of the token read from `N` is more than `max_N`, or if the value of `state` has reached `max_state`.

Example 1 shows the actor written in Java:

Example 1.

```
package ptolemy.actor.lib;

import ptolemy.actor.*;
```

```
import ptolemy.kernel.CompositeEntity;
import ptolemy.kernel.util.*;
import ptolemy.data.*;
import ptolemy.data.expr.Parameter;
import ptolemy.graph.Inequality;

public class CharRead TypedAtomicActor {

    public Parameter max_N;
    public Parameter max_state;
    public Parameter endChar;

    public TypedIOPort N;
    public TypedIOPort Data;
    public TypedIOPort Out1;
    public TypedIOPort Out2;

    private Token state;
    private Token _state;

    public CharRead(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException {
        super(container, name);
        max_N = new Parameter(this, "max_N", new IntToken(12));
        max_state = new Parameter(this, "max_state", new IntToken(99));
        endChar = new Parameter(this, "endChar", new CharToken(""));
        N = new TypedIOPort(this, "N", true, false);
        Data = new TypedIOPort(this, "Data", true, false);
        Out1 = new TypedIOPort(this, "Out1", false, true);
        Out2 = new TypedIOPort(this, "Out2", false, true);
    }

    public void initialize() throws IllegalActionException {
        super.initialize();
        _state = new IntToken(0);
    }
    prefire() {
        return N.hasToken();
    }
    fire() {
        _state = new IntToken(state);
        IntToken n = N.getToken();
        if (n.isLessThan(max_N) && _state.isLessThan(max_state)) {
            if (Data.hasTokens(n)) {
                _state.add(n);
                for (int i = 0; i < n.intValue(); i++) {
                   Out2.send(Data.getToken());
                   Out1.send(_state);
                }
            } else {
               // what to do with the value of n?
            }
        }
    }
    postfire() {
        state = _state;
        return true;
```

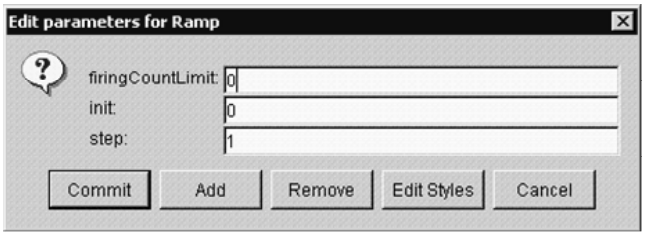


Figure 2.5: An actor's parameters can be set using the parameter window, which appears when right clicking the actor. This actor takes the parameters `init`, `step` and `firingCountLimit`.

```
    }
}
```

2.4.1 Parameters

`CharRead` defines three parameters: `max_N`, `max_state` and `endChar`.

The actor parameters can be set by right clicking the actor in the graphic editor Vergil. Figure 2.5 shows the window for setting the parameters which appears on right clicking the actor in the model.

The parameters are defined as public members of type `Parameter`:

```
public Parameter max_N;
public Parameter max_state;
public Parameter endChar;
```

They are instantiated in the actor constructor. The Parameter constructor take three arguments, a reference to `this`, the parameter name as a String and a default value.

```
max_N = new Parameter(this, "max_N", new IntToken(12));
max_state = new Parameter(this, "max_state", new IntToken(99));
endChar = new Parameter(this, "endChar", new CharToken(" "));
```

2.4.2 Ports

Actors receive and send tokens through their *ports*. A port can either have exactly one channel (single port), or it can have an arbitrary number of channels (multi port).

Ptolemy ports are single ports by default, but they can be set to of type multi port by invoking their `setMultiPort()` method with the argument `true`.

The `CharRead` actor contains two input and two output ports. The constructor of a port object `TypedIOPort` takes four arguments, a reference to *this*, a String containing the ports name and two booleans specifying whether the port is an input or an output port. For input ports those booleans are `true` and `false`, for output `false` and `true`:

```
public TypedIOPort N;
public TypedIOPort Data;
public TypedIOPort Out1;
public TypedIOPort Out2;
...
public CharRead( ... ) {
    ...
    N = new TypedIOPort(this, "N", true, false);
    Data = new TypedIOPort(this, "Data", true, false);
    Out1 = new TypedIOPort(this, "Out1", false, true);
    Out2 = new TypedIOPort(this, "Out2", false, true);
}
```

2.4.3 Tokens

Actors communicate by exchanging *tokens*, which can be defined as atomic pieces of data.

The Ptolemy II, `Token` classes provide a type system with primitive types such as `IntToken`, `DoubleToken`, `BooleanToken`, etc. as well as 'composite' types such as `ArrayToken`, `MatrixToken`. The base class of the token hierarchy is the `Token` class.

Although it is possible to use Java built-in primitive types or Objects for defining variables, we chose to represent the `state` variable by an `IntToken`.

2.4.4 Initialization

When starting the simulation of a model, every actor's `initialize()` method is invoked by the director. This method initializes the actor's global state variables. The `CharRead`'s `initialize()` method instantiates a new `IntToken` object and assigns it to `_state`.

```
public void initialize() throws IllegalActionException {
    super.initialize();
```

```
        _state = new IntToken(0);
    }
```

2.4.5 Split phase firing

The firing of a Ptolemy II actor is divided into three phases:

- Exactly one invocation of its `prefire()` method.
- Any number of invocations of its `fire()` method
- At most one invocation of its `postfire()` method.

2.4.6 Prefire

The `prefire()` method is the only method which is invoked exactly once per iteration. It returns a boolean that indicates to the director whether the actor wishes for firing to proceed.

The `CharRead` actor's `prefire()` method checks whether there is a token available at the input port (without consuming it) and returns `true` if there is one.

```
public boolean prefire() {
    return N.hasToken();
}
```

2.4.7 Fire

The `fire()` method is the main point of execution and is generally responsible for reading inputs and producing outputs. It may also read the current parameter values and the output may depend on them.

Some domains perform a fix-point iteration by calling `fire()` a number of times while each time starting from the original state but consuming new tokens from the input ports. Thus, the `fire()` method should not update the actor's persistent state. Instead, that should be done in the `postfire()` method, which we will discuss in the next section.

The `fire()` method in the `CharRead` actor gets a token from the input port and assigns it to `n`. If the conditions $n < max\_N$ and $\_state < max\_state$ are met, *fire()* checks for availability of tokens at the Data port. If there are at least n tokens available, the outputs are produced. In case there are not enough tokens available,

there are several reasonable solutions of what to do. We will focus on this point in the following section.

```
fire() {
    _state = new IntToken(state);
    IntToken n = N.getToken();
    if (n.isLessThan(max_N) && _state.isLessThan(max_state)) {
        if (Data.hasTokens(n)) {
            _state.add(n);
            for (int i = 0; i < n.intValue(); i++) {
               Out2.send(Data.getToken());
               Out1.send(_state);
            }
        } else {
           // what to do with the value of n?
        }
    }
}
```

2.4.8 Postfire

The `postfire()` method has two tasks: updating persistent state and returning a boolean whether the execution of an actor is complete.

The `CharRead` `postfire()` method assigns the temporary `_state` to the persistent `state` and returns `true`.

```
postfire() {
    state = _state;
    return true;
}
```

2.5 Issues in actor design

Writing atomic actors in Java is not a trivial matter, since it takes a certain knowledge of the Ptolemy II API and of course the Java language itself. Especially when writing domain polymorphic actors, there is a certain risk for creating bugs or unintended behavior, since the actor designer has to be aware of the assumptions which the different domains make about the actors.

Let us illustrate those issues by the example of the `CharReader` actor:

In its `fire()` method, the control structures governing the execution of the action are nested because of the dependency between the value on one port and the number of

tokens to be read from the other. Also, it is important to note that this code defines a way to deal with the eventuality that we have read a token from port N, but did not find a sufficient number of tokens on port Data to start executing. In that case, we need to decide what to do with the value read from N: Do we discard it, or store it for the next time fire is called, in case there is no new token on the N port (in which case we also need to modify the initial condition of the fire method testing for presence of this token)?

The important point here is that possibly the right way of answering this question depends on the model of computation that this actor will be embedded into (and that may be unknown to the author of this actor). Furthermore, the entire question may simply be of no concern to the person writing the actor, and it only distracts from specifying the actual functionality. Finally, the need to write these control structures and possibly manage additional state introduces new ways of making mistakes without contributing to the functionality that a user wants to specify.

Example 2 shows the firing methods of the `CharRead` actor. The lines which do not really concern the actor semantics itself but are necessary for matching the actor to the Ptolemy API are highlighted in bold text:

Example 2.

```
prefire() {
    return N.hasToken();
}
fire() {
     _state = new IntToken(state);
    int n = N.getToken();
    if (n.isLessThan(max_N) && _state.isLessThan(max_state)) {
        if (Data.hasTokens(n)) {
            _state.add(n);
            for (int i = 0; i < n; i++) {
                Out2.putToken(Data.getToken());
                Out1.putToken(_sum);
            }
        } else {
            // what to do with the value of n?
        }
    }
}
postfire() {
  sum = _sum;
}
```

Another drawback following from the example above, is that there is hardly any use for Ptolemy actors written in Java outside the Ptolemy context, since they consist to a big part of code which is specific to the Ptolemy II API.

It would be convenient to have a more abstract representation of atomic actors, which is not specific to the Ptolemy API. This would combine the advantage of a clearer and less error-prone actor definition with an increased re-usability of the designed actors in other contexts than Ptolemy II.

This was the motivation to create CAL, an actor language for specifying data flow actors. The next chapter will show how the `CharRead` actor looks like in CAL, and it will give an introduction to writing actors in CAL.

Chapter 3

CAL

CAL [9] is a textual language for writing data-flow actors which was created as a part of the Ptolemy II project at UC Berkeley. This chapter explains purpose and features of the CAL language, and it provides a brief tutorial to the CAL syntax and semantics.

3.1 Purpose and features of the CAL language

CAL is a domain-specific language for defining the functionality of actors. Its goal is to provide a concise high-level description of an actor, which insulates the actor behavior from the semantics of a specific runtime platform, such as Ptolemy II.

CAL is not intended as a full-fledged programming language, but to be embedded into a richer environment. The language itself does not specify a strict semantics for all the constraints of an actor, such as for example the type system. Neither does it provide communications mechanisms or scheduling schemes for the computational model in which the actor is executed. Instead, it leaves those issues to the designer of the application in which the actor is used.

3.2 Advantages in using CAL for actor definition

In order to show the advantages of CAL, we use the example of the `CharRead` actor again. Here is the `CharRead` actor written in CAL:

Example 3.

actor $CharRead\,(Integer\ max\_N,\ Integer\ max\_state,\ String\ endChar)$
$Integer\ N,\ String\ data \implies Integer\ Out1,\ String\ Out2$:

```
Integer state := 0;
action N : [n], data : [d] repeat n ⟹ Out1 : [sum], Out2 : [d] repeat n
  guard n < max_N, state < max_state
  do
    state := state + step;
  endaction
endactor
```

This version is obviously much shorter and looks significantly more elegant then the Java version, but that is not the deciding point. The main advantage of CAL is that it insulates the author of an actor to deal with the Ptolemy specific constraints which are not really concerning the actors semantics, such as deciding what to do with read tokens if the actor shows not to be firable, keeping a temporary copy of the actor state, how to separate the firing into `prefire()`, `fire()` and `postfire()`, and so on.

Let us now summarize the benefits and drawbacks of CAL:

- *Portability and re-usability of actors:* Actors written in CAL can be used in the context of any actor based simulation platform, provided the existence of a compiler or interpreter, which translates the actor definition in CAL to the platform's API. If a platform's API is modified such that its interface to the actors changes, it is sufficient to modify the CAL compiler/interpreter and recompile the actor library instead of re-writing the whole library.
- *Simplicity of actor design and error prevention:* When writing an actor for a platform like Ptolemy, the author of an actor typically has to bring a certain knowledge about the platforms API. Furthermore, it is hard to avoid that an unexperienced actor author may write erroneous actors, because of not being aware of all the platform-specific constraints. A well written CAL compiler could thus reduce the entrance barrier in actor design as well as the risk for errors, because it could take care of the platform specific issues for the user.
- *Readability and maintainability:* Actors written for a simulation platform in an all-purpose programming language tend to be long and their behavior in certain situations hard to extract from the code. Thus, they need to be very carefully documented, such that a person other than the actors author can reconstruct the actor's intended behavior. CAL offers a compact, clear and precise semantics, which is tailored to the constraints of actor design and thus facilitates readability and maintainability of the actor library.
- *Information extraction for model compilation:* Another, maybe less obvious advantage of specifying an actor in CAL is, that it allows to perform analysis on an actor definition, which is hardly possible with an actor written in Java. In chapter 8 we will explain why information extraction from actors could be useful for further work in model compilation.

- *Cost of writing a compiler:* Writing a compiler for a language as complex as CAL is a nontrivial task, and it takes a lot of workforce to build one from scratch. The CAL compiler for Ptolemy II described in this work, was designed with a focus on ease of retargetability, and intends to be a framework for the design of further compilers to other Java-platforms. A later chapter will focus of the design of this framework.
- *Reduced execution speed of generated actors:* Although an actor defined in CAL looks very compact and clear, it is hardly possible to write a compiler, whose generated code can compete with handwritten code in terms of compactness and execution speed. Depending on the time constraints and importance of the execution speed of a platform, this may or may not be a serious drawback.

3.3 Another simple example of a CAL actor

To get a flavor of the the CAL language, we will start with a quick example using the `Ramp` actor, which is slightly simpler than the `CharRead` actor shown in the last section. The `Ramp` actor produces a sequence of Tokens, where the first token's value is `init` and the value of each following Token is increased by `step`. The actor has one input port which triggers its execution. Example 4 shows the CAL code of the `Ramp` actor.

Example 4.

actor $Ramp\ (Integer\ init,\ Integer\ step)\ Integer\ In \Longrightarrow Integer\ Out$:
  $Integer\ state = init$;
  action $[c] \Longrightarrow [state]$ do
    $state := state + step$;
  endaction
endactor

3.3.1 Actor header

The first line of the `Ramp` actor is called *actor header*. The actor header defines the actor's interface to the model, such as its name, parameters and ports, and opens the *actor scope*. The keyword `endactor` in the last line closes the actor scope again:

actor $Ramp\ (Integer\ init,\ Integer\ step)\ Integer\ In \Longrightarrow Integer\ Out$:
  $...\ this\ is\ the\ Actor\ Scope\ ...$
endactor

Now let us take a closer look at the actor definition:

- The actor definition starts by the `actor` keyword, followed by the actor name `Ramp`.
- The actor name is followed by a pair of braces, which contain the *parameter declarations*. Parameters are constants, whose values can be set by the user through the simulation environment. A parameter definition consists of a type and a parameter name. The `Ramp` actor takes one parameter `k` of type `integer`.
- The last part of the actor definition is formed by the *port declarations*. The arrow separates the *input port declarations* on the left hand side from the *output port declarations* of the right hand side. Both sides have the same syntax and an actor can have any number (including zero) of input and output ports. Each port has a type and a name. The `Ramp` actor has one input port `In` and one output port `Out`.

A double dot marks the end of the actor definition and the beginning of the actor scope. Sometimes it is convenient to break the actor definition into two lines, if it gets too long otherwise. Note that the CAL syntax completely ignores linebreaks.

3.3.2 State variable declarations

Actors have state, which they keep between two firings. State variables in CAL are defined inside the actor scope, but outside the actions. The ramp actor has one state variable `state`, which is defined in the following line:

$Integer\ state = init;$

3.3.3 Action definition

As already mentioned in an earlier section, a CAL actor can have one ore several *actions*. Actions are defined within the actor scope, and each action defines a scope itself, which contains any number of *statements*.

The *Scale* actor has only one action, and its header looks like the following:

action $[c] \implies [state]$ do

It consists of the the `action` keyword, the *port pattern* and the `begin` keyword.

The port pattern contains an arrow which separates the *binding patterns* from the *bound patterns*.

The *binding patterns* define how many tokens the action should read from each input port, and they introduce new variable names, which are bound to the incoming tokens. The binding pattern `[c]` in our example defines, that the action should read one token from the input port `In`, and binds the token read from `In` to the name `c`. If there is no token available at `In`, the action cannot fire. Since binding patterns are very important to the expressiveness of CAL we will explain them more in general in a separate section. For now it is enough to understand the pattern in our example.

The *bound patterns* define how many tokens the action should produce at each output port, and what values the output tokens should take. In our example, the actions bound pattern is `[state]`, which defines the value of the token produced at the output port to be the value of the `state` variable. Bound patterns will be explained more in detail in the next section.

The actor scope contains a list of *statements*, which will be explained in detail in the next section. The `Ramp` actors only action has one statement

$$state := state + step\,;$$

which adds the value of the `step` parameter to the variable `state`.

The example of the `Ramp` actor gave us an impression what the CAL language looks like and showed the skeleton of a CAL actor - the *actor definition* and the *action definitions* - but it does not demonstrate all the capabilities of the language. The next section will describe the introduced constructs more in detail and will introduce more language features.

3.4 A short tutorial to the CAL language

This section provides a short tutorial to actor design in CAL, and introduces the key constructs of the CAL language. A complete documentation of the language syntax and semantics is given in [9].

3.4.1 The structure of an actor

The following example 5 gives a simplified framework of a CAL actor with two actions and introduces names for the grammatical elements. It will help the reader to keep the big picture when specific language elements are described more in detail in the following sections of this chapter.

Example 5.

actor $actorname\ (parameter\ decls)\ input\ port\ decls \Longrightarrow output\ port\ decls$

```
...
(initialization) statements
...
action inputpatterns ⟹ outputexpressions
   guard guard conditions
   var declarations
   do
      (action) statements
   endaction
   ...
   action...
     do
        ...
     endaction
     ...
   endactor
```

3.4.2 Data types

As mentioned previously, CAL is designed to be embedded in a host environment or host language. CAL thus does not provide an own type system, and unstructured types such as `Boolean`, `Integer`, `Double` have to be imported from the host environment. So the choice of these types, as well as the operations on them is left to the environment. In the context of this work, the host environment is the Ptolemy II platform. The Ptolemy type system was already briefly discussed in the last chapter.

3.4.3 Data structures

CAL provides a syntax for a number of built-in data structures, such as *unstructured types*, *tuples*, *comprehensions* and *closures*.

Unstructured types are simple variables, which contain one value of a type like `Boolean`, `Integer`, `Double`, `Complex`. How already mentioned in the last section, the choice of unstructured types is left to the host environment.

Tuples define composite data structures, which are simply collections of member variables. The member variables can be of different types. Example 6 shows the definition of a tuple, containing three variables of types `Integer`, `Double` and `String`.

Example 6 (Tuples).

$[Integer,\ \text{boolean}\ ,\ String]\ myTuple\ =\ [7,\ false,\ 'hello']\ ;$

Comprehensions are parametric types and CAL provides three kinds of built-in comprehensions: *Set*, *Map* and *List*. A *Set* is an unordered set of variables, a *List* is is a Set

whose elements are ordered, and a *Map* is a Set of *key-value* pairs, where *key* and *value* are variables of any primitive type.

All those comprehension types can be iteratively constructed by *generators* and *filters*, and the syntax and semantics is very similar in all three cases:

$\{\, expressions \,:\, generators,\ filters \,\} forSets$
$[\, expressions \,:\, generators,\ filters \,] forLists$
$map\{\, mappings \,:\, generators,\ filters \,\} forMaps$

Let us explain those constructs for a *Set* by the following examples:

Example 7 (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.

The first *Set comprehension* $\{1, 2, 3\}$ contains no generators and no filters but three expressions.

The second definition's comprehension has one expression $2 * a$, and one generator **for** a **in** $\{1, 2, 3\}$. The generator loops through each element of the Set $\{1, 2, 3\}\}$, assigns the current element to a, evaluates the expression $2 * a$ for the current a and puts the result of the expression evaluation into the generated Set.

The third example contains two generators and a filter $b > 2 * a$. The generator evaluation works the same with two generators as with one, the second generator's evaluation loop is simply nested in the first one's. For each pair of a and b generated by the generators, the filter condition is applied, and if it is met, the expression is evaluated for the current a and b.

List definitions have the same syntax as Set definitions, except that they have square brackets instead of curly brackets, and that the order of the elements is relevant.

Map comprehensions work similarly, but they construct *Map* objects, whose elements are *mappings*. The following example shows the use of a Map comprehensions:

Example 8 (Map comprehensions). The following map comprehension
map$\{a \longrightarrow 2 * a :$ **for** a **in** $\{1, 2, 3\}\}$
creates the map $\{1 \longrightarrow 2, 2 \longrightarrow 4, 3 \longrightarrow 6\}$.

The last composite data structure, the *closures* will be explained in section 3.4.10.

3.4.4 Parameter definitions

The *parameter definitions* define constants whose scope is the actor scope, and which can be set by the user through the host application. CAL does not specify how the

setting of the parameters should be implemented by the application, but it allows to define a default value for the parameter, which the user may or may not change. The actor header in example 9 specifies a parameter of type `integer` and sets its initial value to 1.

3.4.5 Input and output port declarations

The *port declarations* in the *action header* define the number of input and output ports, their names and their types. CAL distinguishes between *single ports* which have one *channel*, and *multi ports* which have any number of *channels* (including zero). A multi port is defined by adding the `multi` keyword to the port declaration, if `multi` is omitted, the port is defined as a *single port*. Example 9 shows the actor header of an actor which contains one input multi port and one output single port, both of type double.

Example 9 (Parameter and port declarations).

actor $ParallelToSerial\ (Integer\ MaxChannelNumber\ =\ 1)$
  multi $Double\ ParallelIn\ \Longrightarrow\ Double\ SerialOut$

Tokens consumption from the input ports and token production at the output ports are specified by the *input patterns* and *output expressions*, which we will focus on in the following sections.

3.4.6 Input patterns

The *input patterns* are on the left hand side of the action header's *port pattern* and have the following purposes:

- Defining the number of tokens consumed by the action
- Posing a condition for firability of the action depending on token availability
- Binding tokens or sequences of tokens to variables

Let us start by describing the single port case. Imagine an action which contains the binding pattern $[a, b, c]$ for one of its input ports. This action can only fire if there are three or more Tokens at this input port. When the action fires, the value of the first incoming token at this port is bound to a, the second token to b and the third one to c. The statements in the action scope can now refer to the values of the first three incoming tokens by using the names a, b and c. Binding patterns can thus be considered as a special kind of variable declarations with a special syntax.

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 binds a, b and c again to the first three input tokens, and s to a *List* representing the rest of the input sequence. Tokens four and five at the input sequence can now be referred to as $s[0]$ and $s[1]$, etc.

Example 10 (Single port input 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 the List $[3, 4]$.

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.

In the multi port case things are a little bit different: Since a multi port can have any number of channels, each of them corresponding to a sequence of incoming tokens, a simple variable is not enough to keep one token out of each stream. The solution is to use Maps and *channel selectors* for selecting a particular subset of the input port's channels. There are four different selectors:

- at followed by an integer i selects the i-th channel of the input port. The pattern binds tokens to (unstructured) variables.
- $at*$ followed by a collection c of integers selects all the channels of the input port whose members are elements of c. The pattern binds tokens to *Maps* which all have the key set c.
- all selects all the channels of the input port. The pattern binds tokens to *Maps*.
- any selects only those channels which match the pattern. The pattern binds tokens to *Maps* which all have the set of the matching channels as their key set.

Now let us explain these selectors by some examples:

Example 11 (Multi port input patterns). Assume there is a multi port with four channels, and the following incoming sequences of integer tokens:

Channel	input Sequence
0	$[3, 4, 5]$
1	$[2, 4, 6, 8]$
2	$[9]$
3	$[0, 1]$

The pattern $[a, b \| c]$ *at* 1 binds a to 2, b to 4 and c to $[6, 8]$.
The pattern $[a, b]$ *at* 2 does not match.

The pattern $[a, b, c]$ *at** $\{0, 1\}$ binds a to the map $\{0 \rightarrow 3, 1 \rightarrow 2\}$, b to the map $\{0 \rightarrow 4, 1 \rightarrow 4\}$, and c to the map $\{0 \rightarrow 5, 1 \rightarrow 6\}$.
The pattern $[a \| b]$ *at** $\{1, 2\}$ binds a to $\{1 \rightarrow 2, 2 \rightarrow 9\}$, and b to a map $\{1 \rightarrow [4, 6, 8], 2 \rightarrow []\}$.
The pattern $[a, b]$ *at** $\{1, 2\}$ does not match.
The pattern $[a]$ *all* binds a to the map $\{0 \rightarrow 3, 1 \rightarrow 2, 2 \rightarrow 9, 3 \rightarrow 0\}$
The pattern $[a \| b]$ *all* binds a to $\{0 \rightarrow 3, 1 \rightarrow 2, 2 \rightarrow 9, 3 \rightarrow 0\}$ and b to a map $\{0 \rightarrow [4, 5], 1 \rightarrow [4, 6, 8], 2 \rightarrow [], 3 \rightarrow [1]\}$.
The pattern $[a, b]$ *all* does not match.
The pattern $[a, b]$ *any* binds a to the map $\{0 \rightarrow 3, 1 \rightarrow 2, 3 \rightarrow 0\}$, and b to $\{0 \rightarrow 4, 1 \rightarrow 4, 3 \rightarrow 1\}$.
The pattern $[a, b, c, d \| e]$ *any* binds a to the map $\{1 \rightarrow 2\}$, b to $\{1 \rightarrow 4\}$, c to $\{1 \rightarrow 6\}$, d to $\{1 \rightarrow 8\}$ and b to $\{1 \rightarrow []\}$.
The pattern $[a, b, c, d, e]$ *any* does not match.

For each port there can be one corresponding binding pattern in an action header. The binding patterns can be labeled by *port tags* by using the following syntax: port name:[binding pattern]. If the patterns are not labeled, CAL assumes that they have the same order as the input port definitions. If one pattern is labeled, all of the patterns have to be labeled. The following example 12 illustrates the use of port tags:

Example 12 (Port tags). Assume there is an actor with the following input port declarations:

$Integer\ Select,$ multi $Double\ In \Longrightarrow Double\ Out$

Then the following three input patterns are equivalent:

- [a, b] at i, [i]
- Select:[a, b] at i, In:[i]
- In:[i], Select:[a, b] at i

While the following two patterns are not valid:

- [i], [a, b] at i
- [a, b] at i, In:[i]

3.4.7 Guards, var clauses and action matching

A significant part of the expressiveness of CAL comes from the way how actions are chosen for firing. An Action can only be fired, if it matches the current input in the

current state. The parts of an action definition which are considered during an action matching are the following:

- The input patterns
- The local variable declaration in the `var`-clause
- The boolean expressions in the `guard`-clause

We already focused on the input patterns and how they pose conditions for firability of an action in earlier sections.
The *var*-clause allows definition of local variables, whose scopes are the scope of the action they are defined in. They are initialized before proceeding the *guard*-clause.

The *guard*-clause allows to define a set of additional conditions for *firability* of an action. Those conditions have the form of boolean expressions, and they may reference variables bound in the input patterns as well as variables defined in the *var* - clause.
Example 13 shows the use of a *guard* - and a *var* - clause in the action clause of the `RightShift` actor. The purpose of the *map comprehension* will be explained in the following subsection.

Example 13.

actor $RightShift\ ()\ Double\ InputBus,\ Integer\ Shift \implies$ multi $\ Double\ OutputBus$:
  action $[a]\ all,\ [i] \implies [b]\ all$
    guard $\ i > 0$
    var $\ map\{Integer,\ Double\}\ b =$
      $map\{\ k \mapsto$ if $k < i$ then 0 else $a[k-i]$ end : for $Integer\ k$ in dom $a\}$
    do
    endaction
  endactor

The `RightShift` actor outputs 'connects' each channel k of `InputBus` to channel $k + i$ of `OutputBus`, where i is a token read from `Shift`. The first i channels of `OutputBus` are 'stuffed' by 0. Note that the keyword *dom* followed by a map is the key set of the map.

3.4.8 Output expressions

The *output expressions* are on the right hand side of the arrow, and they define what tokens the action produces at the output ports. A single port expects a *List* of (unstructured) tokens. A multi port expects either a *List* of *Maps* or a *Map* with *Lists* as values. The *Map*'s keys are channel indices in both cases. In the *List* of *Map* case, the output expressions have the same syntax as the input patterns and they can also have *selectors*.

Since there are some problematic cases in the *List* of *Map* representation, we will not focus more in detail on the semantics of output expressions for multi ports. For the

interested reader it is recommend to consult the actual release of the CAL reference manual [9].

The output expression in example 13 uses the *List* of *Map* notation. The map is defined in the *Var*-clause.

3.4.9 Expressions and Statements

Almost every grammatical structure in CAL is either an *expression* or a *statement*.

Expressions in CAL are side-effect-free and strictly typed. CAL provides several kinds of expressions, here are some examples:

- *Literals*, such as 12, 3.141, 'hello'
- *Identifier*, such as variable names, port names, etc.
- *List*, *Set*, *Map* comprehensions
- *If - then - else* - expressions, as in example 13
- The *actor definition* itself
- *Lambda expressions* (will be explained in the next section)

Statements may change the state of an actor, and actions are executed as a sequence of statements. CAL provides among others the following kinds of Statements:

- *Assignment* statements
- *Flow control* statements such as *if- then- else-* statements , *while-* statements, *foreach-* statements
- *exec-* statement (will be explained in a following section)

A complete overview over all the expressions and statements is given in [9].

3.4.10 Functional closures

Closures are objects which encapsulate pieces of code and have an own scope. A *functional closure* result from evaluating a *lambda expression*. Lambda expressions are parameterized expressions, which may be assigned to function variables. The *application* of a functional closure is an expression as well.

Example 14. A function variable `addFunction` is declared and assigned the value of a function closure, which defines two parameters and returns their sum:

$[Double,\ Double \longrightarrow Double]\ addFunction :=$
    lambda $(Double\ a,\ Double\ b) \longrightarrow Double\ :\ a\ +\ b$ endlambda

The expression

$addFunction(2,\ 5)$

is the *application* of the closure, and it is evaluated to the value 7.

Lambda expressions may refer to variables defined in the surrounding context, as in the following example:

Example 15. The function variable `statePlusA` is assigned a functional closure, which returns the sum of `state` and `a`:

$[Integer \longrightarrow Integer]\ statePlusA :=$
    lambda $(Integer\ a) \longrightarrow Integer\ :\ state\ +\ a$ endlambda

Note that the value of `state` is not changed by the functional closure, since expressions are side-effect-free. The `state` variable is said to be a *free variable* of the closure, and the closure can only be applied within a scope which defines a variable `state`.

3.4.11 Procedural closures

Procedural closures encapsulate a list of *statements* which may change the actor state. The syntax of a procedure definition is similar to the syntax of *functional closures*, but there is an important difference in how they are applied. Since the execution of a procedure is likely to have side effects (as opposed to the application of a function), it cannot be part of the evaluation of an expression so it is a statement. A procedural execution starts by the *exec* keyword. The following example illustrates the behavior of a procedural closure:

Example 16. The procedure variable `addToState` is assigned a procedural closure, which adds the value of `a` to `state`:

$[Integer \longrightarrow Integer]\ addToState :=$
    proc $(Integer\ a)\ :\ state\ =\ state\ +\ a$ endproc

The following statement executes the procedure and thereby increases the value of `state` by 2:

$exec\ addToState(2);$

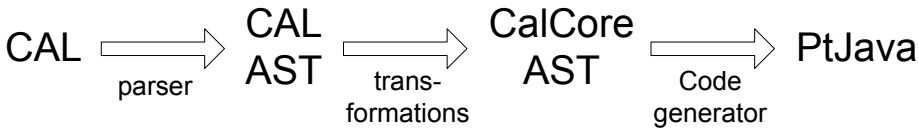
Chapter 4

An overview over the CAL compiler

This work addresses the design and implementation of a code generator for a compiler, which generates actors for Ptolemy II from the actor description language CAL. This chapter gives an overview over the design of the CAL compiler.

On the other hand, this chapter does not provide any general introduction to compiler design and we assume the reader to have some basic knowledge in this area. Compiler design has been a research topic since several decades and there exists a lot of literature about it. The most standard works are [3] and [4]. A well written and very pragmatic introduction is given in [12].

The following figure shows the operational steps performed by the CAL compiler when translating source language *CAL* into the target language *PtJava*:



The CAL compiler is designed as a multi-pass compiler, a *pass* denotes the traversal of the whole source program. Parsing is the first pass, each of the following transformations is done in a separate pass and code generation takes two more passes, as we will see in chapter 7.

A multi-pass compiler is more modular than a single-pass compiler, and it facilitates performing certain context analysis, which a single-pass compiler could not do.

The following sections will explain the figure above more in detail and focus on the

terms it introduces.

4.1 The parser

Parsing is the first pass in the compilation process, and its purpose is to construct the *abstract syntax tree* (CAL AST), a data structure which is a tree representation of the source program. Designing the parser for the CAL compiler was not part of this work. The parser used in the CAL compiler was created with *javacc*, which is a software that facilitates generation of a parser for a language specified by its recursive grammar.

4.2 The CAL AST

The *CAL AST* is a treelike data structure of Java objects. The nodes of this tree represent the reproduction rules of the recursive CAL grammar, and the tree is a complete description of the source program.

Constructing an abstract syntax tree representation of the source program brings the benefit, that each of the following passes can operate on one and the same data structure which is easy to transform and to annotate with context information.

Abstract syntax trees are explained in detail in [3], [4] and [12].

4.3 AST Transformations

Before generating the target code, the CAL compiler performs a set of transformations on the abstract syntax tree. The transformations have the following purposes:

- *Annotation of the AST* with information needed by the following transformations or the code generator.
- *Replacing grammatical structures by more basic ones*, in order to get to a subset of the grammar, which is semantically rich enough to support all constructs of the full language, but which is easier to implement code generation for.

The following chapter will give an overview over the transformations performed on the AST. It will explain their purpose and describe what the AST looks like after the transformations.

4.4 The CalCore AST

CalCore AST is what we call the AST after the transformations. *CalCore* is supposed to be a minimal subset of CAL, which still supports all the necessary constructs to describe an actor's semantics. A complete documentation of the CalCore sub-language is given in [9].

4.5 Code Generation

Code generation is the last step in the compilation process. The code generator takes the CalCore AST data structure as input and generates a stream of characters representing the actor in Java. Design and implementation of the code generator was the main task of this work. Chapter 6 will explain the structure of the generated code and chapter 7 will focus on design and some particular implementation details of the code generator.

4.6 PtJava

The target language of our CAL compiler is Java. Since generated actors are supposed to match the Ptolemy API we will refer to the target language as *PtJava*. The *Ptolemy II* API was described in chapter 2, and the structure of the generated code will be explained in chapter 6.

Chapter 5

Transformations on the AST

This chapter lists the transformations which the current CAL compiler performs on the AST, and briefly explains their purpose. The transformations are described in the same order as they are executed during compilation.

5.1 Unary and binary operation removal

The first two transformations performed during compilation address removal of unary and binary operations, such as

- The *dom* operator (unary)
- The *not* operator (unary)
- Additional operations $+$, $-$ (binary)
- Boolean operations: and, or (binary)
- Comparisons: $=$, $<$, $>$, $<=$, $>=$ (binary)

Each of these operations is replaced by a corresponding *function application*, where the function to apply is included from a library. Replacing operators by function applications has the following advantages:

- It eliminates the *unary operation* and the *binary operation* construct from the language, which reduces the required input for implementing or re-targeting the code generator.

- It allows code generation for those operations, without providing a static type-checking.

The second issue might require some explanation: The evaluation of unary and binary operations depends on the types of their arguments. Two `integers` for example have a different semantics for the + operator than two `Maps`. Because of that, translating binary operations into Java expressions requires a static type checking which determines the operand types at compile time. Since a static type checking for CAL does not exist yet, the compiler transforms the + operator into an application of the global `Plus()` function, which checks for the operand types at runtime.

5.2 Variable annotator

CAL distinguishes between variables which are *assignable*, and variables which are not. Non-assignable variables are called *constants*. *Parameters* in function applications for example are *constants*, since once they have been initialized at instantiation they cannot be re-assigned.

Another property which variables can have or not, is *mutability*. Mutability is only relevant for composite data structures such as *Sets*, *Lists* and *Maps*, and it denotes, whether single elements of the composite structure can be re-assigned or not. Sequences are represented as immutable *Lists*, whereas user defined *Lists* usually are mutable.

Assignability and mutablity has a certain impact on how variables are declared and initialized, as we will see in the implementation chapter. Because of that, the compiler maintains an *environment*, which contains a *binding entry* with the variables properties for each variable. The purpose of the *variable annotator*, is to annotate each *variable reference* occurring in the CAL code with a reference to their binding entry. When the compiler has to generate the code for the variable reference, it can find the variables binding entry through the annotation and check for the variables assignability and mutability properties.

5.3 Port tagging transformer and input pattern canonicalizer

As already explained in the CAL chapter, input patterns can occur with or without tags. The port tagging transformer transforms all the patterns into the *tagged* notation, by adding a tag to every port pattern which does not have one yet.

The *input pattern canonicalizer* transforms every input pattern to a pattern which simply binds the incoming sequence to a sequence variable. The variables bound in the

original pattern are declared and initialized in the `var`-clause instead. Let us illustrate this by the following example:

Example 17. Imagine there is an action which looks as follows:

action $[a, b \parallel c] \implies [d]$ do
$\ldots$
endaction

After having passed the input pattern canonicalizer and the port tagger, the action looks as follows:

action $Input : [\ \parallel \$G0] \implies Output : [d]$
guard $\$G1$
var
boolean $\$G1 = \$available(\$G0,\ 2)$,
$Integer\ a = \text{if}\ \$G1\ \text{then}\ \$G0[0]\ \text{else}\ null\ \text{endif}$,
$Integer\ b = \text{if}\ \$G1\ \text{then}\ \$G0[1]\ \text{else}\ null\ \text{endif}$,
$Set[Integer]\ c = \text{if}\ \$G1\ \text{then}\ \$subsequence(\$G0,\ 2)\ \text{else}\ null\ \text{endif}$,
do
$\ldots$
endaction

The second form is perfectly valid CAL syntax, and although the first version looks much more elegant, the user could write the actor directly in the second form. The purpose of this transformation is, again, to simplify the grammar of the CalCore AST, such that the code generator needs to deal with less grammatical constructs.

5.4 Dependency annotator and sorter

There are two places in a CAL*action header*, where variables can be defined: in the *input patterns* and in the `var`-clause. As we saw in the last section, definitions in the *input patterns* are moved to the var clause by the *annotator*.

Variable definitions can depend on each other, so it matters in what order they are performed. In the input patterns and the `var`-clause, variable may be defined in any arbitrary order. When generating the Java code, the variable definitions have to be sorted. The *dependency annotator* annotates dependencies in the variable declarations, and the *sorter* puts them in the right order.

The following example gives an example of a set of variable declarations before and after sorting them:

Example 18. Assume a `var`-clause of an action to contain the following variable declarations:

$Double\ b\ =\ a[k];$
$Integer\ k\ =\ i\ +\ 2;$
$Integer\ i\ =\ 0;$
$[Integer]\ a\ =\ [1,\ 2,\ 3,\ 5,\ 7,\ 11,\ 13]$

The first variable b cannot be initialized, since a and k are still undefined at this time. Variable k in the second line cannot be initialized as well, since i is not defined until the next line. After the sorter has run, the variables are in the right order:

$Integer\ i\ =\ 0;$
$Integer\ k\ =\ i\ +\ 2;$
$[Integer]\ a\ =\ [1,\ 2,\ 3,\ 5,\ 7,\ 11,\ 13]$
$Double\ b\ =\ a[k];$

5.5 Transforming composite expressions and statements

CAL provides a set of composite expressions and statements, such as:

- if-then-else expression
- comprehension generator expression
- while statement
- foreach statement
- if-then-else statement

Each of those expressions and statements has an own transformer, which replaces the corresponding construct by a function invocation, or a nesting of function invocations and closure definitions.

Chapter 6

Structure and behaviour of the generated actors

This chapter discusses the issues in designing the structure of the generated code and explains the design decisions that were made. Furthermore, it describes what generated actors look like, how they interact with the Ptolemy API and the runtime infrastructure that had to be added to Ptolemy II in order to support execution of the generated actors.

6.1 Design considerations

The following list describes the considerations and optimization criterions which we considered as being the most important for the design of the code generator. The issues are listed in order of decreasing importance and they will help to motivate the ideas and design decisions described in the following sections:

1. Making retargetability of the code generator as simple as possible
2. Simplify the implementation of the code generator for Ptolemy II
3. Execution Speed of the generated actor

While there seemed to be a positive correlation between the optimizatioin of the first two issues, their optimization seemed to antagonize the optimization of the execution speed and vice versa. In most of the trade-off design decisions taken, more weight was put on retargetability than on speed constraints.

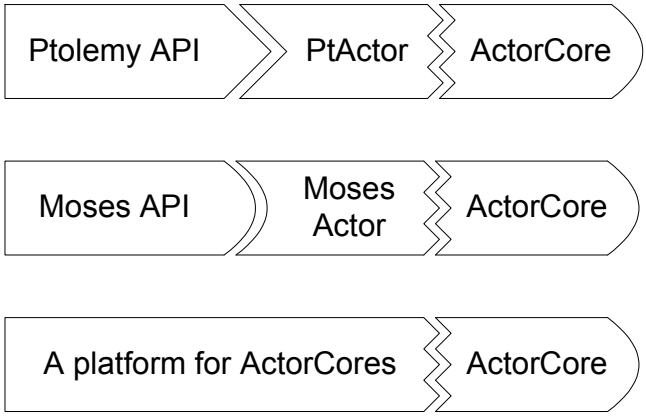


Figure 6.1: An ActorCore can be re-used for several platforms if a corresponding adapter exists, and the costs for retargeting the code generator are reduced. Some platforms might even have a statical interface to the ActorCore and use it without an adapter.

6.2 Generic and specific part of the generated actor

In order to make retargetability of the code generator as easy as possible, we decided to separate the target code into a generic, platform independent part and a Ptolemy II specific part. Retargeting the code generator only requires changes in the Ptolemy specific part, and keeping the specific part as small as possible reduces the amount of code that has to be read, understood and replaced for retargeting of the code generator.

Figure 6.1 illustrates how the generic and the specific code are encapsulated into two separate objects *ActorCore* and *PtActor*, and how *PtActor* works as an adapter between the *ActorCore* and the Ptolemy API. As a side effect, the *ActorCore* can be re-used for other platforms than Ptolemy II, such as *Moses* (described in [1]).

For some platforms it is even imaginable that the platform specific part of the actor might be the same for every actor and thus could be implemented as a static library instead of being generated. In the figure this is named as *platform for actor cores*.

In the next sections we will see how the *PtActor* interacts with the *ActorCore* at run-time, and how they communicate by using a *Factory* object which is provided by the *runtime environment*.

6.3 Interactions with the runtime environment

This section explains how the *PtActor* interacts with the *ActorCore* at runtime, and how they communicate by using a *Factory* object which is part of the *runtime environment*.

6.3.1 What is the runtime environment?

The *runtime environment* is a library, which extends the Ptolemy API by a set of classes that enable the execution of generated actors. The purpose of this package is to put as much functionality as possible into this static library, instead of generating those functionalities into each generated PtActor. This simplifies both implementation and retargetability of the code generator, since it is easier to write functionality into a static library than to write a generator which generates this functionality.

The *runtime environment* provides the following elements:

- A *Factory* for creation of Ptolemy specific objects in the ActorCore and for communication between the actor objects
- A port wrapper for emulating random access to the Ptolemy input ports, which only allow sequencial access to their tokens
- Classes for representation of CAL*variables*
- A *change listener*, which handles state shadowing of the variables
- A set of *global functions*, which the actors can access
- *Interfaces* through which the ActorCore can access those ptolemy specific structures

In this chapter we will describe the interactions between the *runtime environment* and the generated actors on a rather abstract level. The next chapter will explain the components of the runtime environment more in detail.

6.3.2 Passing ports and parameters through the factory

Even though the *ActorCore* is generic, it needs to have some kind of access to objects which are specific to the Ptolemy II platform. It needs to get input tokens from the Ptolemy input ports and parameters, and it has to send tokens to Ptolemy output ports. But since the *ActorCore* is generic, it may not have references to any Ptolemy specific object.

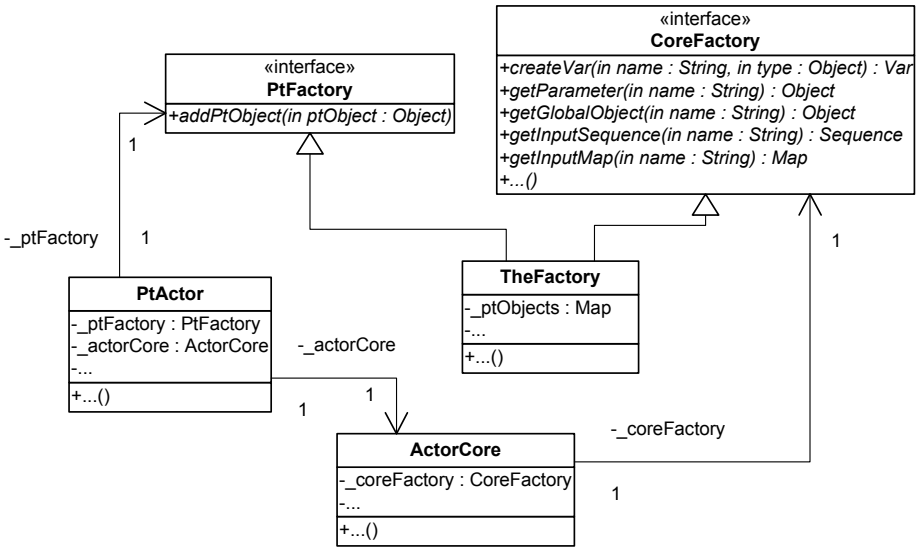


Figure 6.2: The *PtActor* pushes Ptolemy ports and parameters into *TheFactory* by calling its *addPtObject* method. The *ActorCore* can access thoses objects through the methods of the *CoreFactory* interface

This problem is solved by the *Factory* object, provided by the runtime environment. Figure 6.2 illustrates in a simplified UML scheme, how the *Factory* collaborates with the actors at runtime. *TheFactory* implements two interfaces:

- *PtFactory* provides a method *addPtObject*, which allows the *PtActor* to pass Ptolemy specific objects to *TheFactory* and saving them into the map `_ptObjects`.
- *CoreFactory* provides a set of methods for accessing the objects saved in the map. Those methods all return objects of not ptolemy specific types, such as *Map*, *Sequence*, *Object*. We will see in the next chapter how those objects are wrapping the corresponding Ptolemy specific objects.

6.3.3 Creating CAL variables using the factory

Ptolemy II actors use *Token* objects to represent variables. The *ActorCore* may not reference those objects directly in order not to lose its generality. CAL variables in the ActorCore are thus represented by *PtVar* objects which wrap the platform specific Token objects, and the ActorCore accesses those objects through the *Var* interface which they implement. Since the ActorCore may not instantiate the Ptolemy specific PtVar

objects itself, it uses the `createVar()` method of the *Factory* to create those variables:

```
Var variableName = _coreFactory.createVar(type, initValue);
```

6.3.4 Actor and runtime environment instantiation

The *PtActor* is instantiated by the Ptolemy II application, when the user places it into a model. The *PtActor* then creates the objects of the runtime environment *ActorCore*.

Figure 6.3 shows the order in which the objects are created:

1. The user starts the simulation of the model. The application first calls the actor's `initialize()` method.
2. *PtActor* pushes a parameter into the factory by calling the factory's `addToPtObjects` method.
3. *PtActor* pushes a parameter into the factory by calling the factory's `addToPtObjects` method.
4. *PtActor* calls *ActorCore*'s *initialize()* method.
5. *ActorCore* pulls the parameter from the factory by invoking `getParameter()`.
6. *ActorCore* gets an output port wrapper object by `getOutput()`.
7. *ActorCore* creates a state variable by using the factory's `createVar()` method.
8. The factory instantiates a *PtVar* object, which it returns to the *ActorCore*.

6.3.5 Actor initialization

The `initialize()` method of the *PtActor* pushes the Ptolemy port and parameter objects into the factory by invoking the factory's `addPtObject()` method. This was already explained in section 6.3.

After having passed all the Ptolemy specific objects to the factory, the *PtActor* calls the `initialize()` method of the *ActorCore*, which pulls the Ptolemy objects from the factory.

Figure 6.4 gives an example of a runtime initialization:

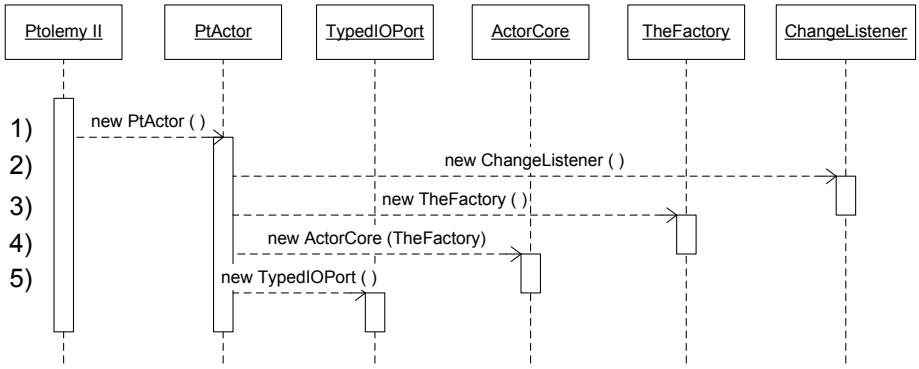


Figure 6.3: The *PtActor* pushes Ptolemy ports and parameters into *TheFactory* by calling its *addPtObject* method. The *ActorCore* can access thoses objects through the methods of the *CoreFactory* interface

1. The user, usually by dragging the actor into the model window in Vergil. The *PtActor* constructor is invoked by the Ptolemy application.
2. *PtActor* instantiates the *ChangeListener* object.
3. *PtActor* instantiates *TheFactory* by passing it the change listener.
4. *PtActor* instantiates the *ActorCore* by passing it the factory object.
5. *PtActor* instantiates a *TypedIOPort* object. Of course, an actor can have any number of ports, and there could be any number of parameters instantiations as well.

6.4 The structure of the ActorCore

The last section focused on the instantiation of the *ActorCore* and its collaboration with the *PtActor* and the environment. This section explains the structure of a generated *ActorCore* more in detail. Figure 6.5 shows the *ActorCore* and lists the members and methods, which appear in every *ActorCore*:

6.4.1 Variable representation

As already mentioned, CAL variables are represented by *PtVar* objects which implement the *Var* interface. The *Var* interface provides two public methods:

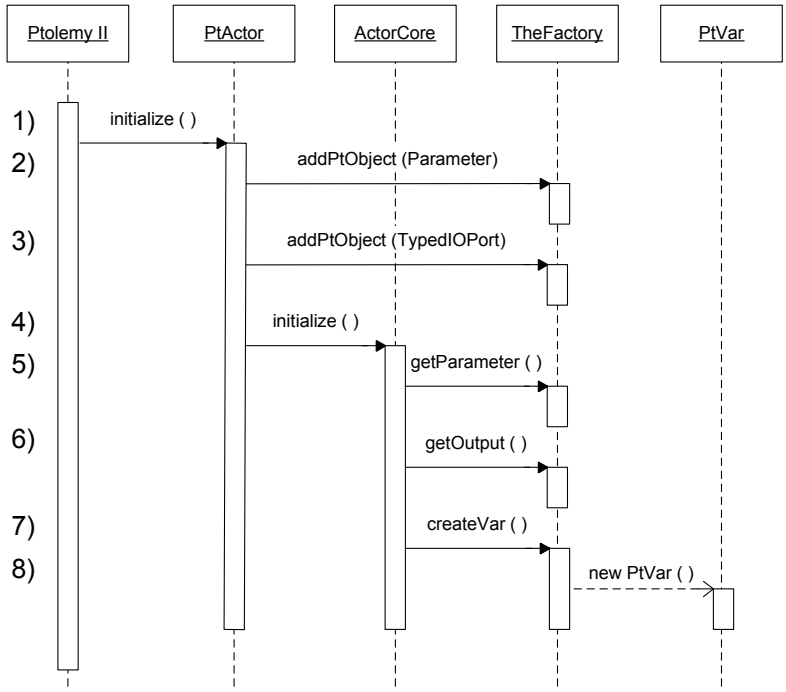


Figure 6.4: The *PtActor* pushes Ptolemy ports and parameters into *TheFactory* by calling its *addPtObject* method. The *ActorCore* can access thoses objects through the methods of the *CoreFactory* interface

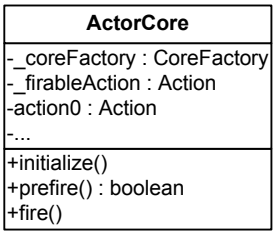


Figure 6.5: The minimal member infrastructure of a generated *ActorCore*

- `assign(Object o)`, which assigns the Object `o` to the variable
- `value()`, which returns the Object contained in the variable

The next chapter will focus on the *Var* interface and the *PtVar* class in more detail.

6.4.2 Action Objects

Actions in CAL have their own *scope*. Variables defined locally in the action scope cannot be accessed from outside the action. In order to implement action scoping, the code executed by the *action* is encapsulated into an *inner class* of the ActorCore, which implements the *Action* interface. Variables defined locally in the CAL action scope are thus defined as members ot the inner Action class.

Each Action Object provides the following methods:

- `prefire()`, which evaluates the input patterns, the *var* clause and the *guard* conditions and returns a `boolean` whether the action matches or not.
- `fire()`, which executes the CAL statements in the statement block of the action and sends resulting output tokens to an output port wrapper

Actions are instantiated in the ActorCore's `prefire()` method, which will be explained in the following section.

6.4.3 The prefire method

Each ActorCore contains a `prefire()` method, which returns a `boolean` whether at least one of the actions can fire or not. It assigns a *reference to the firable action* to the member `_firableAction`. The code of the `prefire()` method is rather straightforward and for an actor with two actions it looks as follows:

```
public boolean prefire() throws IllegalActionException {

    action0 = new Action0();
    if (action0.prefire()) {
        _firableAction = action0;
        return true;
    }

    action1 = new Action1();
    if (action1.prefire()) {
        _firableAction = action1;
        return true;
    }

    _firableAction = null;
    return false;
}
```

The reason why action objects have to be newly instantiated in each fire cycle will be explained in section 6.4.8.

6.4.4 The fire method

The `fire()` method of the ActorCore is the same for every actor and looks as follows:

```
public void fire() {
    _firableAction.fire();
}
```

It simply invokes the `fire()` method of the actor that was stored as firable in `prefire()`.

6.4.5 Closure Objects

Functional closures in CAL, just like actions, define their own scope. Thus, closures in the generated actor are represented by inner classes, which implement the *Closure* interface. Since closure declarations in CAL are expressions, they can appear almost anywhere in the actor definition. As a result, *Closure* objects can be inner classes of an *Action*, another *Closure* or the *ActorCore* itself, as shown in figure 6.6.

The *Closure* interface defines one function `apply(Object args)`, whose argument can be an object or an array of objects. The `apply(Object args)` method

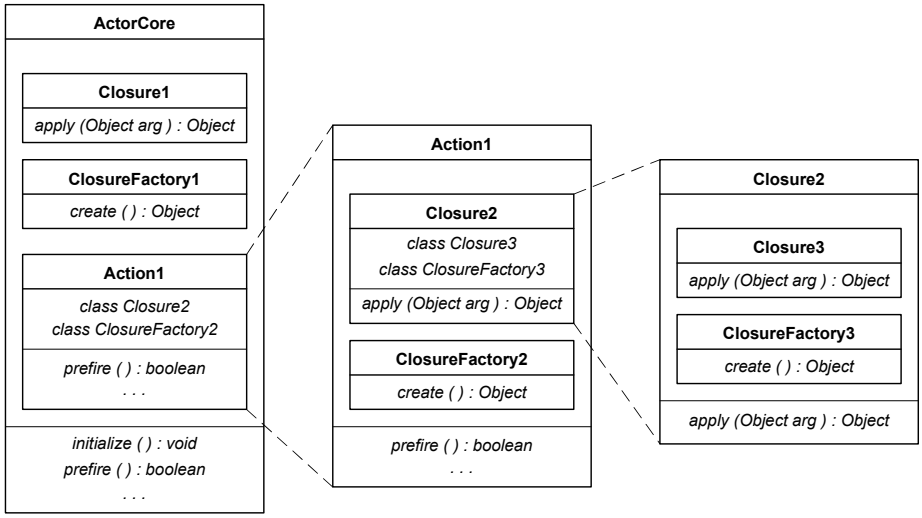


Figure 6.6: Closures can be inner classes of *Action* objects, other *Closure* objects or the *ActorCore*

extracts the parameters from the passed array, evaluates the lambda expression and finally returns the evaluation result. The following example shows the structure of a generated *Closure* object, which takes two parameters:

```
public class lambda1 implements Closure {

Object __a;
Object __b;

public Object apply(Object arg) {
    Object[] argArray = (Object[])arg;
    __a = argArray[0];
    __b = argArray[1];
    ...
    // evaluate expression of __a and __b
    ...
    return evaluationResult;
}
};
```

6.4.6 Closure instantiation

Closure objects are newly instantiated each time before their `apply()` method is invoked. This might seem like a waste of resources, but it is absolutely necessary to do

so, as we will see later in this chapter.

There is one problem that arises from the fact that closures have to be re-instantiated before their application: Since function variables can be assigned any closure object, the code generator cannot know at compile time what closure the function variable will refer to at runtime, so it cannot simply generate the code for closure instantiation to the place in the *ActorCore*, where the function is applied.

To solve this problem, the code generator creates *ClosureFactory* objects, which are inner classes of the *ActorCore* as well, and instantiate their corresponding closure object. When a closure is assigned to a function variable in CAL, the corresponding Java code assigns a *ClosureFactory* to the variable object. When the function is applied, first a new instance of the closure is produced by the factory's `create()` method, then the closures `apply()` method is invoked.

In the following example a function variable __add is instantiated and later assigned a *ClosureFactory*. Finally, the assigned *Closure* is instantiated and applied to a and b:

```
Var __add = _coreFactory.createVar(null, null);
...
__add.assign(new ClosureFactory1());
...
__tuple5[0] = a;
__tuple5[1] = b;
Object aPlusb = ((ClosureFactory)__add).create().apply(__tuple5);
...
```

6.4.7 Procedural closures

Procedural closures are realized exactly the same way as functional closures. Procedural closures implement the *Procedure* interface, which defines a funtion `exec(Object arg)` for starting procedure execution. Procedure objects are instatiated by their corrensponding *ProcedureFactory* object.

6.4.8 Why actions and closures need to be re-instantiated

The goal of this section is to explain, why action objects have to be newly instantiated at each firing cycle, and why closure objects need to be instantiated before each application or execution.

The following example shows an actor implementing the *Sieve of Eratosthenes*. The actor produces the sequence of all prime numbers provided that its input is connected to a source which produces the sequence of the natural numbers starting at 2. The source actor could be for example the *Ramp* actor introduced in chapter 2, with parameters $init = 2$ and $step = 1$.

Example 19.

```
actor PrimeSieve () Integer Input ⟹ Integer Output :

  [Integer ⟶ boolean ] filter := lambda (Integer n) ⟶ boolean : false endlambda

  [Integer, Integer ⟶ boolean ] divides :=
          lambda (Integer a, Integer b) ⟶ boolean : b mod a = 0 endlambda

  action [a] ⟹ []
    guard filter(a)
  endaction

  action [a] ⟹ [a]
    guard not filter(a)
    var [Integer ⟶ boolean ] f = filter
    do
      filter := lambda (Integer n) ⟶ boolean : f(n) or divides(a, n) endlambda
    endaction

  endactor
```

The first two initialization statements declare and initialize two function variables:

- *filter* is initialized to refer to a closure, which simply returns *false*. The function application *filter(x)* will thus return *false* for any *x*.
- *divides* is initialized to a closure, which checks whether its first argument divides the second argument.

Now the actor performs a sequence of firings, where it has two actions to chose from. Remember that the input sequence is assumed to be the natural numbers starting with 2, so $a = 2$ in the first firing, $a = 3$ in the second, and so on:

- The first action does not match, since *filter(2)* is *false*
- The second action matches, and *filter* is assigned a new closure which evaluates *false or divides(2, n)*
- In the second firing again, the first action does not match, since *false or divides(2, 3)* is *false*

- The second action matches, and *filter* is assigned a new closure which evaluates *false or divides(2, n) or divides(3, n)*
- In the third firing, the first action matches, and *filter* is not changed.
- In the fourth firing again, the first action does not match.
- The second action matches, and the expression in the *filter* closure keeps growing to *false or divides(2, n) or divides(3, n) or divides(5, n)*

The expressions *false or divides(0, n) or divides(1, n) or ...* contain several application of the same closure. Since those closures are partly evaluated with different parameters (their first arguments are bound to $0, 1, 2, \ldots$) we need to keep track somehow of those different sets of bound parameters. Since parameters are members of the lambda closure, the simplest way to solve this problem is to re-instantiate the closure every time it is applied.

The expression illustrates as well, why we re-instantiate the action object in each firing cycle. If we would not do so, there would be only one instance of the variable a. Each firing cycle would change its value to the value of the next incoming token, and would hereby change the arguments previously passed to the *divides* functions. The variable *filter* would thus refer to *false or divides(k, n) or divides(k, n) or ...* after $k + 1$ firings. Re-instantiation of the action allows to easily keep track of multiple values for a.

6.4.9 Global functions

As explained in the last chapter, many caltrop features such as binary operations, generation and indexing of comprehensions, foreach-statements and so on, are replaced during the transformations by global functions and procedures.
Those global closure objects are provided by the *Factory*, and the actor core can access them by using the access methods specified in the *CoreFactory* interface. The following example shows how a generated *ActorCore* gets the *Plus* method from the factory and applies it to the variables a and b:

```
// ActorCore class members
private Object __Plus;
...
// in the initialize method
__Plus = _coreFactory.getGlobalObject("Plus");
...
// in an action or a lambda closure
__tuple5[0] = __a;
__tuple5[1] = __b;
Var sum = ((ClosureFactory)__Plus).create().apply(__tuple5);
...
```

The function variable is defined as a member of *ActorCore*. It is initialized in the `initialize()` method and applied in the same way as a user defined function. The *global variable annotator* introduced in the *transformations* chapter helps the code generator to include only those functions into the generated actor, which are really needed.

6.5 The Ptolemy II specific part of a generated actor

For reasons of retargetablilty, it was one of our design goals to make the Ptolemy specific part of the actor, which we will refer to as *PtActor*, as simple as possible. Thus, the structure of the *PtActor* is rather straightforward and its firing methods look pretty much the same for every actor. This chapter explains the design of the *PtActor*.

6.5.1 Construction and Initialization

The constructor of the *PtActor* instantiates the following objects:

- the *PtVarChangeListener*
- *TheFactory*
- the *ActorCore*
- the actor ports
- the actor parameters

The `initialize()` method pushes the Ptolemy port and parameter objects into the factory by invoking the factory's `addPtObject()` method. This was already explained in section 6.3.

6.5.2 Ports and Parameters

The *PtActor* defines, instantiates and configures the actor's ports and parameters. This is done in exactly the same way as in a handwritten actor, which was already explained in chapter 2.

6.5.3 Firing methods

The firability of CAL actions usually depend on the availability of tokens at the input ports, and it thus seems reasonable to check for token availability in the *PtActor*'s

`prefire()` method. This can be achieved by simply invoking the *ActorCore*'s prefire method, since this method does check for availability of tokens.

However, there is a Ptolemy specific problem with this solution which has a severe drawback: In the CT domain of Ptolemy there are no tokens available at the actors input ports at the time their `prefire()` methods are invoked. Thus, models containing any actor which checks for token availability in its *prefire* method cannot execute in CT.

In order to make the generated actors usable in the CT domain, we decided to introduce a *compiler flag*, wich allows the user to control whether the `prefire()` method of the *ActorCore* should be invoked from the *PtActor*'s `prefire()` or its `fire()` method.

The following table shows for the two possible values of the compiler flag, how the *PtActor* invokes the firing methods of the *ActorCore*:

PtActor method	$CTFlag = 1$	$CtFlag = 0$
prefire()	`...` `return true;`	`...` `return _actorCore.prefire();`

fire()	`...`
	`if (_actorCore.prefire()) {`
	`  _actorCore.fire();`
	`}`
	`return;`
		`...`
		`_actorCore.fire();`
		`return true;`

postfire()	`...` `return true;`	`...` `return true;`

Note that the functions above are not complete. Some lines for exception handling *state shadowing* and other things were omitted for didactical reasons. State shadowing will be explained in the next section.

6.5.4 Variable change listener and state shadowing

In chapter 2 we already explained that the `fire()` method of an actor should not update the persistent state of the actor. In some domains, the actor's `fire()` method is invoked several times, each time starting from the original state. As soon as the desired result is achieved, the actor state is persistently updated by invoking its `postfire()` method.

This *state shadowing* could be implemented by introducing new temporary variables

for the shadowed state, but this would make the ActorCore depend on this particular way of using it. In order to faciltate the *state shadowing* without losing generality of the ActorCore, the *PtActor* contains a member of type *PtVarChangeListener*, whose reference is passed to the factory constructor when the latter is instantiated. Every variable produced by the factory gets a reference to the change listener upon its construction.

Whenever a variable is assigned by using its `apply()` method, the *PtVar* object notifies the *PtVarChangeListener*, which enters its reference into a map. The *PtActors* then uses the following two methods of the change listener, in order to realize the state shadowing:

- `rollBackAll()` loops through all the variables which were noted as changed, and sets them back to the original value. After that, the map of changed variables is cleared. The rollback method is invoked in the beginning of the *PtActor*'s `fire()` mehod.
- `commitAll()` makes all the variable changes permanently and clears the map of changed variables. The commit method is invoked in `postfire()`.

Figure 6.7 shows a simplified example of the instantiation, initialization and a first firing of an actor, which summarizes most of the mechanisms presented by this chapter. The arrows concerning state shadowing are marked by numbers, and we will go through each of them in the following enumeration:

1. The *VarChangeListener* object is instantiated by the *PtActor*.
2. In the beginning of the PtActor's `fire()` method, the change listener's `rollback()` method is invoked. In this case, nothing happens, since no variable was assigned to yet.
3. The *ActorCore* assigns a new value to an earlier defined variable. The variable's `assign()` method notifies the change listener.
4. The *PtActor* `postfire()` method, calls the change listener's `commitAll()` method, in order to commit the variable changes.
5. The change listener's `commitAll()` method invokes each changed *PtVar*'s `commit()` method. In this example, only one variable was changed.

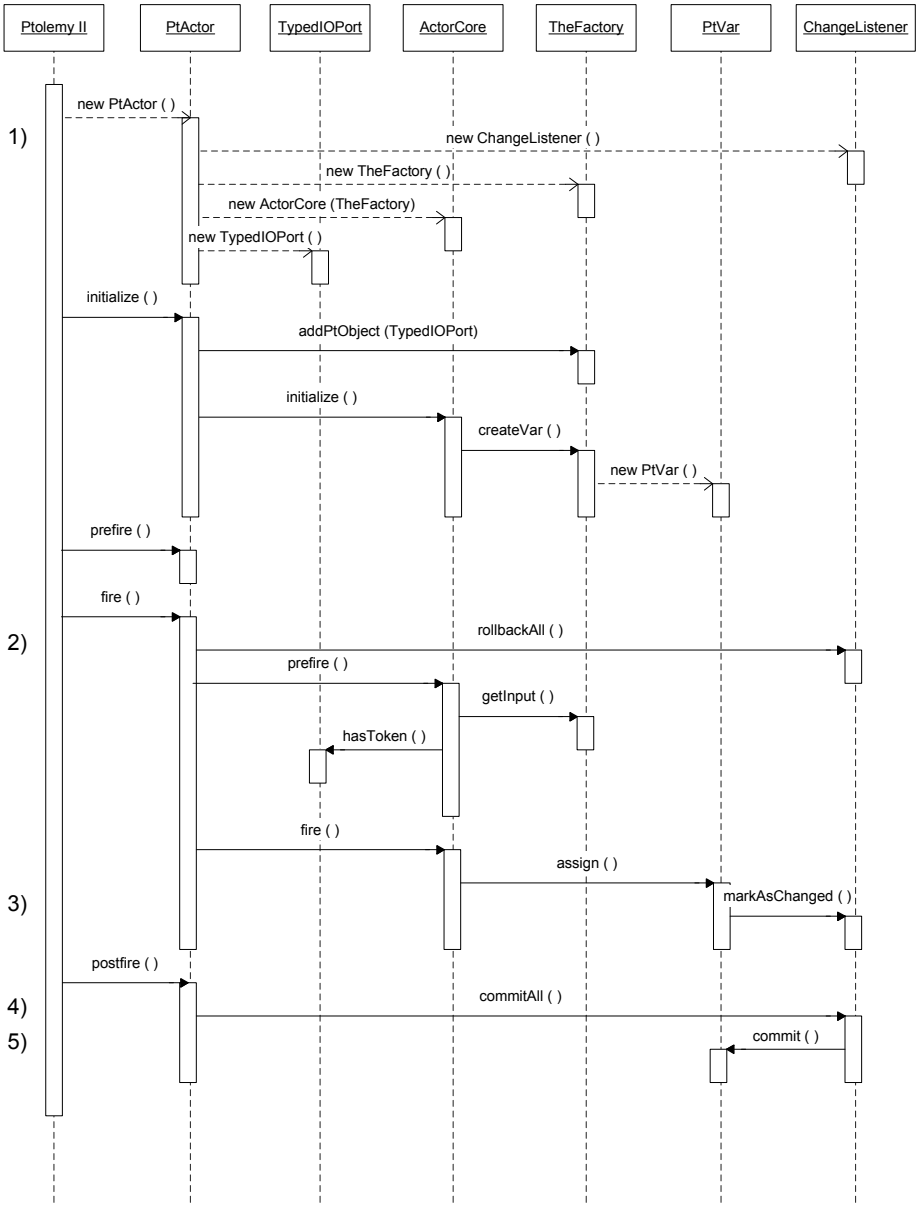


Figure 6.7: A simplified summarizing example of how Ptolemy II application, generated actor and runtime environment interact at runtime. The arrows highlighted by numbers are those which concern *state shadowing*.

Chapter 7

Implementation

7.1 The code generator

This section provides a high-level view on some implementation issues of the code generator. First we will focus on the structure of the code generator, then we will explain how the code generator uses the *visitor pattern* to traverse the input data structure, and finally we will take a look on the `TargetCode` classes, which provide an intermediate representation of the generated code.

7.1.1 Generic and Ptolemy specific code generator

The last chapter explained how generated actors are separated into a generic `ActorCore` and a platform specific `PtActor` class. Since each of those classes can be generated separately, we decided to split up the code generator into a generic and a specific part as well. Figure 7.1 shows the structure of the two-part code generator.

Re-targeting the code generator requires writing a new platform specific part for the code generator, which is illustrated in the figure by the dashed arrows.

7.1.2 The visitor pattern

The input data structure to the code generator, the *CalCore AST* was introduced in section 4.4. The code generator traverses traverses the CalCore AST and generates the corresponding Java code for each node in this tree.

The most straightforward implementation for generating code would thus be to add a code generation method to every AST node, which generates the Java code from the

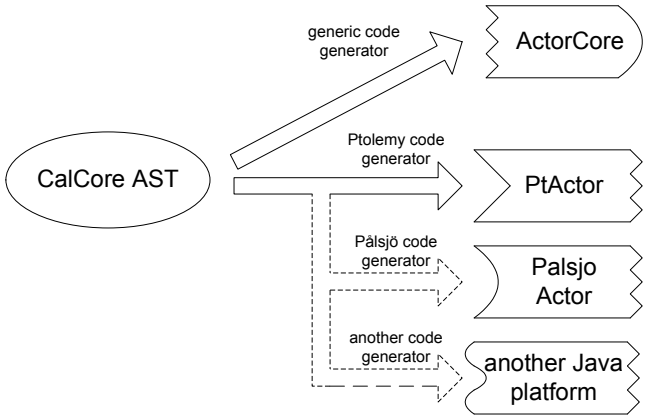


Figure 7.1: The code generator is divided into a *generic code generator* for generation of the `ActorCore`, and a *Ptolemy code generator* which generates the adapter object `PtActor`. The dashed arrows show how the code generator can easily be re-targeted to a new Java platform such as `Pålsjö`, by simply re-targeting the specific part of the code generator.

information contained in the node. The code generator could then simply traverse the tree and invoke each node's code generation method.

But using this approach has a severe drawback: It mixes the AST data structure with the operations performed on it. Remember that the transformers and other code generators have to operate on same AST structure. The described solution would thus require adding one method per transformer and code generator to each AST node, which would cause the AST classes to grow and to become hard to maintain. It would be cleaner to avoid 'polluting' the AST structure with the transformation or code generation operations, which any of the applications performs on it.

The solution to this problem is a design, which [7] refers to as the *visitor pattern*. Figures 7.2 and 7.3 illustrate the implementation of the code generators using the visitor pattern.

Figure 7.2 shows the class structure of the generic and the specific code generator `CoreVisitor` and `PtVisitor`. Both of them extend the class `BasicVisitor`.

Figure 7.3 shows how the pattern works at runtime:

1. The visitor invokes the `accept()` method of the current node (which is *ExprApplication*) by passing it a reference to *this*.
2. The node calls back the visitor's *visitApplication()* method by passing it a reference to *this*.

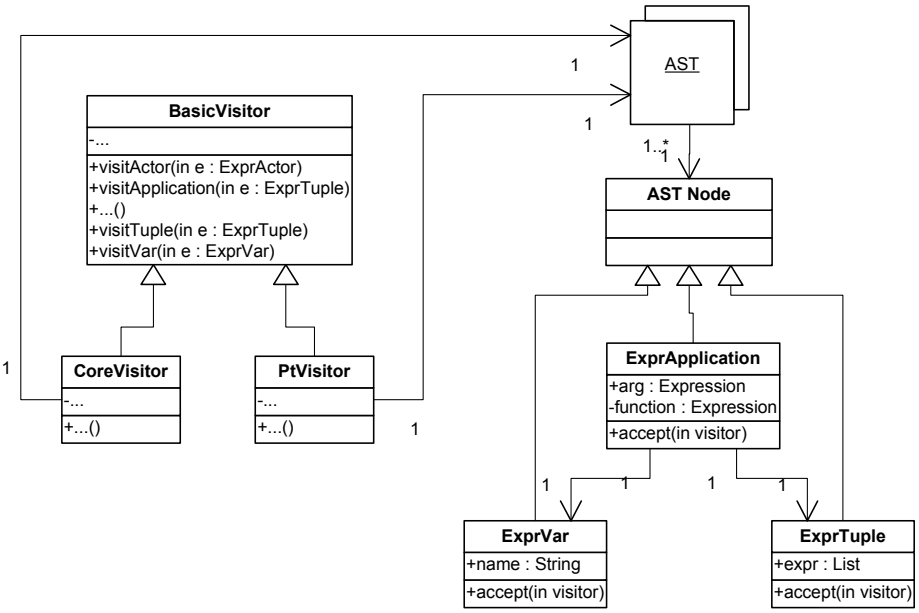


Figure 7.2: Implementation of the code generator using the visitor pattern: The *visit* methods in *BasicVisitor* control the traversal of the AST, but do not perform any operations on the nodes. Each code generator extends this base class and overwrites some or all of its *visit* methods with code generating methods.

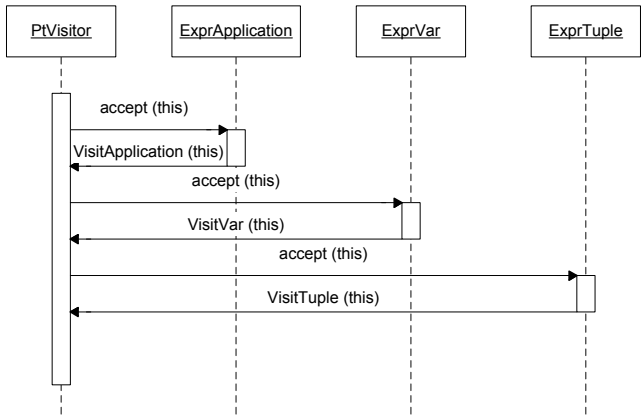


Figure 7.3: The *PtVisitor* visits the *ExprApplication* by invoking its `accept()` method. ExprApplication calls the corresponding visit method in the PtVisitor, which generates the Java code for the function application. While generating the code it evaluates the function's name and parameters by visiting *ExprVar* and *ExprTuple*.

3. The visitor 'knows' that the descendants of the *ExprApplication* are *ExprVar* and *ExprTuple* and visits both of those nodes by calling their `accept()` method.

The `BasicVisitor` traverses the AST without generating any code or changing the tree. In order to perform operations on the visited nodes, the inherited classes *PtVisitor* and *CoreVisitor* simply have to overwrite the corresponding *visit methods* by methods performing those operations. If a visit method for a node is not overwritten, the code generator will not generate any code for this node, but it will still traverse the descending nodes and eventually generate code for those nodes.

7.1.3 Generating code for expressions

Most of the grammatical elements in CAL are either a *statement* or an *expression*. Every existing CAL statement can be generated into a Java statement or a sequence of Java statements. But CAL expressions do not always translate into a Java expression but some of them translate into a sequence of statements and expressions. This has certain consequences as we will explain in the following example:

Example 20 (Code generation for an ExprApplication). Imagine we want to generate code for an *ExprApplication*, as it is shown in figure 7.2. Note that an ExprApplication is a function invocation and its descendants *ExprVar* and *ExprTuple* are the function name and a tuple containing the arguments passed to the function.

If both the ExprVar and the ExprTuple could be translated into simple Java expressions, the `visitApplication` method of the code generator could be implemented as follows:

```
public void visitApplication(ExprApplication e) {
    e.function.accept(this); // print Java expr for function name
    addToTargetCode("(");
    e.arg.accept(this); // print Java expr for tuple creation
    addToTargetCode(");");
}
```

The reason why this does not work is in the third line: CAL tuples are represented by object arrays in the generated code, and there is no way to instantiate and initialize a Java array in one simple expression. Of course we could define a function somewhere which instantiates a tuple and initializes its values with the passed arguments, but since we would have to provide one function for every possible tuple length, this is not a very reasonable solution.

This problem is solved by passing a String containing the expression evaluation result between the two visit methods using the global state variable `_exprResult`.

The `visitTuple()` method (which is called by the `accept()` method) adds the generated Java statements to the target code but stores the generated Java expression String into `_exprResult`.

The `visitApplication()` method assigns the String containing the tuple expression to `tuple` and then calls the accept method for evaluating the function name. The function name is passed using `_exprResult` again, and in the last line of `visitApplication()`, function and tuple are combined into a Java function application expression which is stored in `_exprResult` again:

```
public void visitApplication(ExprApplication e) {
    e.arg.accept(this); // the invoked accept() method prints
                        // the Java statements for creating
                        // the tuple, f.ex.:
                        //   Object[2] _tuple;
                        //   Object[0] = arg1;
                        //   Object[1] = arg2;
                        // then it assigns the name of
                        // the tuple object[] to _exprResult
                        //   _exprResult = _tuple;
    String tuple = _exprResult;
    e.function.accept(this); // the invoked accept() method
                             // assigns the name of the
                             // function to _exprResult
    String function = _exprResult;
    _exprResult = ... + function + ... + "(" + tuple + ")";
}
```

So let us summarize how this works in general: Visit methods for expression nodes add generated Java statements to the target code but write the resulting Java expression as a String into the `_exprResult` variable. This global variable is available for the visit method of the parent node, and the parent visit method concatenates the String value `_exprResult` with the code it generates for the parent node. The resulting String is then either printed to the target code or written into `_exprResult` again, depending on if the expression is part of a Java statement or another Java expression.

The implementation using the state variable `_exprResult` solves the problems caused by the fact, that CAL expressions can translate into Java statements in the target code. The dots in the last line symbolize that there is some more code in the real version of `visitApplication` which was omitted in order to keep the example as simple as possible.

7.1.4 Variable names

When generating the target code, the code generator may introduce new internal variables in the Java code to store intermediate results and temporary objects. The names

of those variables consist of a prefix and a sequence number.

Java variables representing CAL variables consist of a prefix and the variable name in CAL. The prefix is to avoid name collisions in the generated actor, in case the user defines a CAL variable with a name which is used for compiler generated classes or objects, such as `Action1` or `envChangeListener`.

7.1.5 An intermediate representation of the target code

Chapter 6 explained how the generated code is separated into the actor classes `PtActor` and `ActorCore`. Both of those classes contain a number of methods, and the latter one can even contain an arbitrary complex structure of nested inner classes. Generating code for one AST node often requires adding code to more than one of those methods and inner classes.

As a result of that, the generated code cannot be written sequentially. There has to be an intermediate more treelike representation of the target code, which allows the visitor to add code to a specific method or inner class of the generated actor class.

The `caltrop.codegen.target` package provides such an intermediate representation, which is depicted in the static UML scheme in figure 7.4. Now let us explain the purpose of those different interfaces and classes:

- The `Code` interface provides one simple function `toList()`, which is implemented by all the other classes. The purpose of this function is to convert the target code contained in the class implementing this interface from a tree representation into a list of Strings.
- The `CodeLine` class is basically just a wrapper around a String, which implements the toList() method.
- The `CodeChunk` interface defines two `add()` methods, which are supposed to facilitate adding code to the code in `CodeChunk`. One of those methods takes a `String`, the other one takes any object implementing `Code` as parameter.

There are four classes, which implement the `CodeChunk` interface. They can be composed into a tree structure representing the generated actor with its nested inner classes, methods and members. This tree structure uses `CodeClass` objects as intermediate nodes, since `CodeClass` has a Map `methodsAndClasses` containing any number of references to other objects implementing `CodeChunk`. In [7] this kind of combining objects is called *composite* pattern.

`CodeList` has a member `_theCode` containing a list of `CodeLines` representing the lines of a code block.

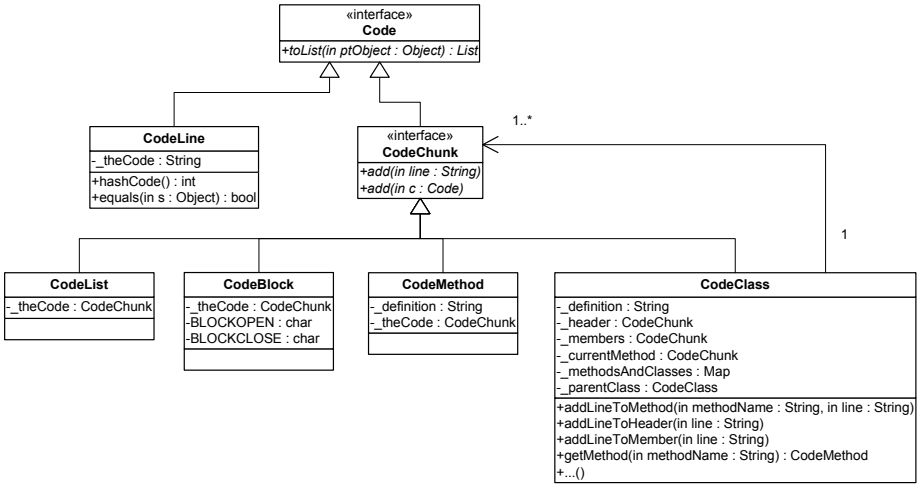


Figure 7.4: The classes of the `target` package can be composed into a tree structure which is an intermediate representation of the generated code.

A `CodeBlock` is almost the same as a `CodeList`, except for including two characters for opening and for closing a block which are added to the list of Strings produced by the `toList()` method.

A `CodeMethod` is basically a `CodeBlock` which contains an additional String representing the method definition.

`CodeMethod` could have been inherited from `CodeBlock` and the latter one from `CodeList` in order to save some writing work. But since the effort for re-writing such a few members is not a matter, we decided not to use inheritance here in order to keep the class hierarchy as flat as possible.

`CodeClass` contains methods for adding code to its class members or to one of its methods. Besides that, it has methods for getting a reference to a `CodeChunk` representing the class members, a certain method or an inner class. Since the method names shown in the UML scheme are quite self explaining, we will not list them and explain them more in detail.

In order to decouple the code generator from the code classes in figure 7.4, the package provides the two classes `PtolemyClassCode` and `CoreClassCode`. Both of them contain a set of methods for constructing the target code structure consisting of the classes in figure 7.4, and for writing code into arbitrary nodes in this structure.

The generic code generator uses the class `CoreClassCode` for accessing the classes shown in figure 7.4. `CoreClassCode` provides the following members and methods:

- A variable _rootClass referring to a CodeClass which is the root class of the tree of nested inner classes. A basic set of methods is added to this root class by the constructor, same as in PtClassCode.
- A variable _currentClass referring to the current CodeClass object in the hierarchy of inner classes.
- A set of addLineTo..() methods in order to add a line of code to a specific method or the class members of the _rootClass object.
- The methods beginNewAction(), beginNewLambda(), beginNewProcedure() which instantiate a new CodeClass object, set its _parentClass to the object refered to by _currentClass, and set _currentClass to the newly instantiated object. Those three methods mainly differ in what String they put into the definition of the instantiated CodeClass.
- A set of addLineToInnerClass..() methods in order to add a line of code to a specific method or the class members of the _currentClass object.
- A set of get..() methods in order to get a reference to a method or the class members of the _currentClass object.
- A method exitInnerClass() which sets _currentClass to its parent CodeClass object.
- And finally there is a printAll() method for printing out the class file to an output stream.

By using the methods of the CoreClassCode object, the generic code generator can add code to the main class and the last added inner class of the target code. This simplifies implementation of the visitor functions and make the code generator easier to understand and to maintain.

PtolemyClassCode is the corresponding facade object for the Ptolemy specific code generator. PtVisitor uses it for having random access to the members of the generated code. The following list gives a short description of those functionalities:

- PtolemyClassCode contains a map for storing CodeMethod objects.
- The constructor of PtolemyClassCode initializes this map by a set of methods which are common to every ptolemy specific actor. The method definitions are hard coded as constants in PtolemyClassCode.
- The class provides a set of addTo..() methods in order to add a line of code to a specific method or the class members.
- The class contains a set of get..() methods in order to get a reference to a specific CodeMethod or the CodeList containing the class members.

- There is a `printAll()` method for printing out the class file to an output stream. This method uses the `toList()` methods of the contained class objects.

The Ptolemy specific part of a generated actor does not contain any inner classes. Thus the `PtolemyClassCode` does not provide any methods for adding inner classes to the target class.

Chapter 8

Conclusions

This chapter summarizes the achievements of this work and explains opportunities for further work in the context of the CAL code generation.

8.1 Achievements

This work covers the design and implementation of a code generator for the actor language CAL. We summarize the achievements as follows:

1. The code generator can generate code for the full-fledged CAL language supporting grammatical constructs such as:
   (a) higher-order function closures
   (b) procedural closures
   (c) Set/List/Map-comrehensions
   (d) input port patterns
   (e) regular action selectors (was not explained in chapter 3)
   (f) the 'usual' control structures such as `if then else`, `foreach`, etc.
   (g) ...
2. The code generator is easily retargetable to other Java platforms, due to its strict decoupling between generic and Ptolemy specific part.
3. The code generator is a powerful framework for creating code generators for other target languages such as C. The code generator has already been re-targeted to the Pålsjö platform [6] at LTH Lund, Sweden, where generated actors are used for controlling a mobile robot.

4. The generated actors are surprisingly efficient, even if the primary optimization goal was to maintain modularity between specific and generic part of the generated code instead of optimizing for speed.

5. The strict decoupling of generic and specific part of the target code makes generated actors reusable for other platforms then Ptolemy II.

8.2 Further work

Even if the code generator produces surprisingly efficient actors, there are still a number of opportunities for improving their efficiency:

- *Speed optimization for Ptolemy II:* Starting with the current modular code generator, the execution speed of the generated code for the Ptolemy II platform could probably be optimized by using Ptolemy specific assumptions, sacrificing generality for efficiency and implementing special treatment of special cases.
- *Re-targeting to other platforms:* In order to make CAL accessible to a broader community the language could be re-target to further platforms, such as Moses [1] or LegOS.
- *Implementing static type checking:* The current version of the compiler does not provide any static type checking. Type errors result in runtime exceptions, which makes them harder to localize. A static type checking would help to make type errors in actor specifications detectable at compile time.
- *Describe AST transformations in xslt:* The transformations performed on the AST are currently defined in Java, and some of them are very hard to read and understand. A more abstract specification of the transformations would help to make the transformations more flexible and maintainable.

Figure 8.1 shows current and further work in the broader context of CAL and Ptolemy, that build on the results of this work.

There have been efforts to compile Synchronous Data Flow models [10] into C, where the actor specifications are given in Java. But it is hard to analyze Java actors and to extract their SDF specific behavior, since the information relevant for SDF is implicit.

Another approach to compile models into C is to create a transformation which transforms a model consisting of CAL actors into one single actor. This is performed in the following two steps:

1. $network + actors \rightarrow schedule$
2. $network + actors + schedule \rightarrow actor$

The resulting actor can then be compiled into C by using a re-targeted CAL compiler. While the current efforts focus on Cyclostatic Data-flow Models in particular, it is a long-term goal to be able to compile several kinds of models into C.

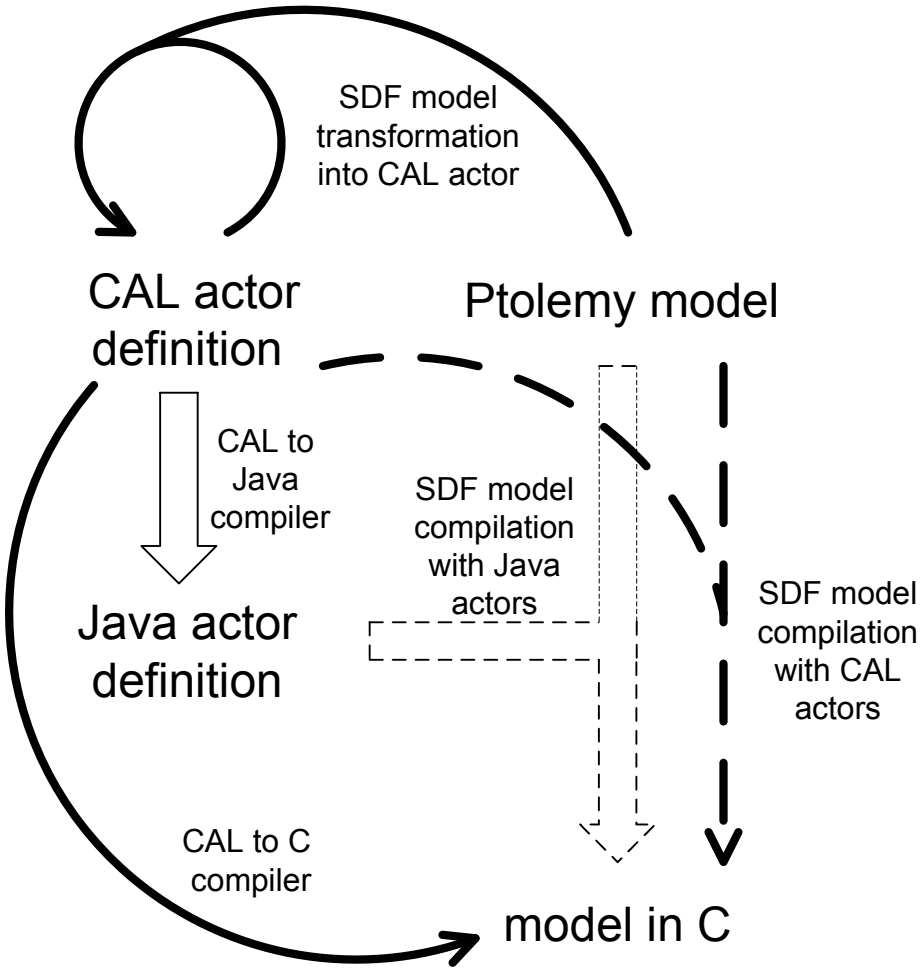


Figure 8.1: The dashed thick arrow shows the efforts to compile models into C by using actor specifications in Java. But extracting explicit information from actors written in Java is very difficult. Another approach of model compilation is to re-target the CAL compiler to C, and to implement an algorithm which transforms an SDF model with actors specified in CAL into a single CAL actor. The CAL actor can then be compiled to C by the CAL to C compiler.

Bibliography

[1] The Moses Project. Computer Engineering and Communications Laboratory, ETH Zurich ($http://www.tik.ee.ethz.ch/\sim moses$).

[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] Jeffrey D.Ullman Alfred V.Aho, Ravi Sethi. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, 1998.

[5] John Davis, 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, EECS, University of California, Berkeley, March 2001.

[6] Johan Eker and Anders Blomdell. A flexible interactive environment for embedded controllers. *IFAC Control Engineering Practice*, (2), February 2000.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

[9] Jörn W. Janneck and Johan Eker. Cal language report. Technical memorandum, EECS, University of California, Berkeley, 2002.

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

[11] 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.

[12] David A. Watt and Deryck F. Brown. *Programming language processors in Java*. Prentice Hall, 2000.