ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Static Analysis of Actor Networks

Diploma Thesis
presented by

Ernesto Wandeler
ETH Zürich, Switzerland

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

Prof. Dr. Walter Schaufelberger
Automatic Control Laboratory
Swiss Federal Institute of Technology, Zurich

March 2003

Acknowledgement

Abstract

In this thesis we present a new interface theory based approach to static analysis of actor systems. We first introduce a new interface theory, which is based on Interface Automata [5] and which is capable of counting with natural numbers. Using this new interface theory, we can capture both, the temporal aspects of an actor interface as well as an actor's token exchange rate. We will show, how to automatically extract this information from actors written in the Cal Actor Language (CAL). We further present a method to capture the interface information as well as the connection information of dataflow system environments into an interface automaton. In our approach, this automaton acts as glue between the actor automata of the system, and by successfully composing all actor automata with it, we can prove interface compatibility of all actors with the system environment. After successful composition, the composition automaton will contain the complete token exchange information of the composite actor system. We will extract this information into a Petri Net, which we then analyse to make statements on boundedness, deadlock as well as on the existence of legal firing sequences for the composite actor system.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Component based design is an approach to software and system engineering, in which new software designs are created by combining pre-existing software components. The ability to check the compatibility of such components, and to detect errors in the composition of components is thereby an important factor for software development productivity and software quality.

Many modern programming languages which make extensive use of component libraries, such as Java or C#, provide component interface compatibility checking on a data type level. By checking whether the data types of arguments in a method call to a component are compatible with the specified interface they ensure a certain level of component compatibility.

Interface theories for component based design [6] take the interface compatibility checking one step further by providing formalisms to specify component interfaces in more detail. Interface Automata [5] is an interface theory, which defines a light-weight formalism that can capture the temporal aspects of software component interfaces. In particular, Interface Automata can capture both, input assumptions on the order in which the methods of a component may be called, as well as output guarantees on the order in which the component calls external methods.

In the Ptolemy project [1], which studies actor based modeling, simulation and design of concurrent, real-time, embedded systems, efforts were made recently, to use Interface Automata to capture the temporal aspects of actors, into what was called the Behavioral Type of an actor [15].

Up to now, the interface automaton of actors and of components in general has to be specified by hand and gets annotated to the source code for later interface compatibility checking, as for example in [3]. However, much of the information contained in an interface automaton is already contained in the source code of a component. Thus, having to specify the interface automaton by hand, introduces a new source for errors into a system design.

In this thesis, we present a solution for automatic extraction of interface information from actors written in the Cal Actor Language (CAL), and we present a strategy, how the extracted information can be used to analyse dataflow actor systems.

1.2 Contributions

We present a new interface theory, which we call Counting Interface Automata (CIA). Counting Interface Automata are based on Interface Automata [5] and are capable of counting with natural numbers. Using CIA, we can capture both, the temporal aspects of an actor interface as well as its token exchange rates.

In order to automatically extract the CIA of a CAL actor, we first develop an intermediate format for actors written in CAL, which we call Calflow. In Calflow the internal data-dependencies of a CAL actor are resolved explicitly and an actor is represented as a set of execution steps in a dataflow graph.

We then present a set of mapping rules, which extract the interface information of a Calflow actor into a CIA. Furthermore, we also present a method to create a CIA of a dataflow system environment, which on one side represents the interface information of the environment, and which on the other side acts as glue between the CIAs of the actors in a system.

With the successful composition of the system environment CIA and all actor CIAs, we can then prove interface compatibility on CIA level, for all actors and the system. The successful composition leads to a new CIA, which contains the complete token exchange information of the composite actor system.

We will show, how to represent the token exchange information of the composite actor system as a Petri Net and we will then use Petri Net methods to make statements on the existence of legal firing sequences, as well as on boundedness and deadlock of the composite actor system.

1.3 Overview

The thesis is subdivided into five parts.

- The first part lays the foundations for the presented work. It describes CAL and presents Calflow.

- In the second part, Counting Interface Automata are defined. Techniques and methods are presented to extract CIAs from actors and models and to use the extracted automata for analysis of actor compositions.

- The third part presents a number of case studies, that show the usage of the techniques and methods which were described in the second part.

- The fourth part contains conclusions and outlooks.

- In the Appendix some detailed information about Calflow is contained, furthermore the implemented software is presented as well as Xchain, a software framework for XML processing, which was developed in connection with this thesis.

1.4 Remarks

1.4.1 List of Abbreviations

Table 1-1: List of abbreviations used in this thesis.

CAL	Cal Actor Language
CIA	Counting Interface Automaton
CSP	Communicating Sequential Processes
CT	Continuous Time
DDF	Dynamic Dataflow
DE	Discrete Events
DF	Dataflow
DOM	Document Object Model
MoC	Model of Computation
MPA	Model Product Automaton
PLU	Property Look-Up
RT	Run-Time
SDF	Synchronous Dataflow
TEA	Token Exchange Automaton
TEPN	Token Exchange Petri Net
URI	Uniform Resource Identifier
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language: Transformation

1.4.2 Typographic Conventions

Code is represented `monospaced`, with keywords highlighted in `monospaced boldface.` Semantic entities are set `monospaced italics` and comments that are not part of the

code are written in *normal italics*. Pseudo-code is represented in *italics*, with keywords highlighted in ***italics boldface***.

Note, that both CAL and Calflow ignore line-breaks and whitespace.

1.4.3 Naming Conventions

Action names, as well as global counter variable names in CIA need to be globally unique. We therefore build them by concatenating actor, port and command/variable names according to the following patterns.

- `ActorName:PortName:CommandName`

- `ActorName:PortName:VariableName.`

Part I
Foundations

Chapter 2

Component Based Design

Component based design is an approach to software and system engineering, in which new software designs are created by combining pre-existing, reusable modules into a model, which provides both glue between the components inside and new functionality to the outside.

2.1 Hierarchical and Heterogeneous Modeling
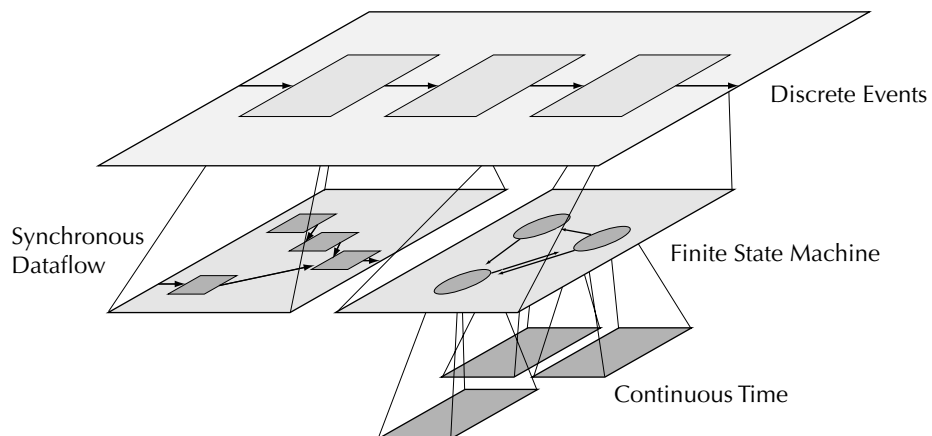
Fig. 2-1: A hierarchical and heterogeneous model.

Embedded systems are often heterogeneous in that they mix different technologies as for example analog and digital electronics. Such systems are often modelled hierarchically, consisting of a number of sub-models, where each of the sub-models may behave differently, to model a particular technology.

2.2 Actor Oriented Modeling

Actor oriented modelling is an approach to systems design, in which entities called actors communicate with each other through ports and communication channels. From the point of view of component based design, actors are the components in actor oriented modelling.

The concept of actors was first introduced in [11] as a mean of modeling distributed knowledge-based algorithms and was also described in [4]. Actors have since then become widely used, especially in embedded systems design, where actor-oriented design is a natural match to the heterogeneous and concurrent nature of embedded systems.

In the context of this work, an *actor* is a computational entity with a well defined component interface. It has *input ports*, *output ports*, *state* and *parameters*. An actor communicates with other actors by sending and receiving atomic pieces of data, called *tokens*, through its ports, along unidirectional connections, called *channels*. Actors are connected through channels, to form *models*.

When an actor is executed it is said to be *fired* and it may consume tokens from its input ports, produce tokens on its output ports and update its internal state, based on consumed tokens, its parameters and its former state.

2.3 Models of Computation

The syntactic structure of an actor-oriented design says little about the semantics of the model. The semantics is largely orthogonal to the syntax, and is determined by a *model of computation* (MoC). A MoC governs the interaction of components in a model. For this, it defines operational rules for executing a model and it also defines the nature of communication between components.

Below, an overview to some models of computation is given. In this thesis we focus on dataflow MoCs, however to get a better feeling of what a MoC is, some other common examples are also presented here.

- *CSP — Communicating Sequential Processes:* In communicating sequential processes models, actors represent concurrently executing processes that communicate by atomic, instantaneous actions called rendezvous[1].

  Rendezvous models are particularly interesting to model applications where

---

[1] Also often called synchronous message passing.

resource sharing is a key element, such as in client-server models, multitasking or multiplexing of hardware resources.

- *CT — Continuous Time:* In continuous time models, the connections between actors represent continuous-time signals and the actors typically specify algebraic or differential equations between these signals. The job of an execution environment is to find at every time step a fix-point that satisfies all relations in a system.

  Continuous time models are usually used to represent the physical parts of systems, which can be modeled using differential equations. In embedded systems these could for example be any mechanical components or analog circuits.

- *DE — Discrete Events:* In discrete event models, actors use timed events for communication, and the connections between actors represent sets of events placed on a time line. An event consists of a value and a time stamp.

  Through the very much physical notion of time in discrete event models, they are an excellent model for describing concurrent hardware. Discrete event models are thus very popular for specifying digital hardware and also for simulating telecommunication systems.

- *DF — Dataflow:* In dataflow models, actor computations are triggered by the availability of input data. Connections between actors represent the flow of data from a producer actor to a consumer actor and are typically buffered with FIFO queues.

  Dataflow models are especially useful to model data-driven processing as in signal processing.

- *SDF — Synchronous Dataflow:* The synchronous dataflow model of computation [13] is a particularly restricted special case of dataflow, where all actors in a model must have constant data consumption and production rates. This leads to the extremely useful property that deadlock and buffer boundedness are decidable and moreover, the scheduling of actor firings as well as the buffer capacity for communication can be computed.

- *DDF — Dynamic Dataflow:* The dynamic dataflow model of computation does not restrict the data consumption and production rates of actors, but properties such as deadlock and boundedness are usually not statically decidable any more.

Chapter 3

Actor Based Modeling in Ptolemy II

3.1 The Ptolemy Project

The Ptolemy project at UC Berkeley studies heterogeneous modelling, simulation, and design of concurrent 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.

A major effort in the Ptolemy project is put in the construction of Ptolemy II, a software system implemented in Java. Ptolemy II is a system-level design environment that supports the modeling, simulation and design of component-based heterogeneous systems.

More information on the Ptolemy project can be found on the project web-page [1], as well as in [8]. Detailed information on the Ptolemy II software can be found in [7].

3.2 Ptolemy Object Model

Ptolemy II takes a component view of design, in which models are constructed as a set of interacting components, called *actors*, and the channels of communication between actors are implemented by so called *receivers*. The receivers themselves are contained in *IOPorts* (input/output ports), which are in turn contained in actors, as shown in Figure 3-1.

In this producer/consumer model of Ptolemy II, communicated data is encapsulated in so called *tokens*. The *producer* actor can deposit tokens into a receiver, and the *consumer* actor can extract tokens from its receiver.

IOPort

Producer
Actor

Consumer
Actor

Receiver
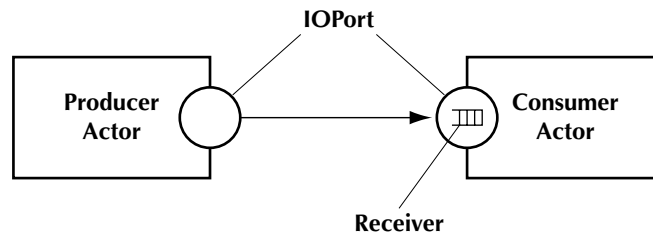
Fig. 3-1: The producer/consumer model in Ptolemy II.

Actors in Ptolemy II are computational entities with a well defined component interface, which communicate through ports and channels. They consume tokens from their input ports and produce tokens on their output ports. Additionally they may have state and parameters.

Aside from assuming a producer/consumer model, the abstract receiver of Ptolemy II, which is defined in a Java interface, makes no further assumptions on the communication process. It does, for example, not determine whether communication between actors is synchronous or asynchronous, nor does it specify buffer capacities or any other details of a receiver.

Instead, these properties are determined by concrete classes that implement the receiver interface. Each of these concrete classes is part of a Ptolemy II *domain*, which is a collection of classes that implement a particular model of computation. In each Ptolemy II domain, the receiver implementation determines the communication protocol, and an object called the *director*, controls the execution of actors. From the point of view of an actor, the director and the receiver form its execution environment and implement the model of computation.

3.3 Composite Actors

In Ptolemy II, when a director is placed in an actor model, that actor model becomes an *opaque composite actor*. To the outside environment, this composite actor is just another actor with a well defined component interface. But inside, it is a composite, executing under the semantics defined by the local director. This concept of composite actors is a key elements for hierarchical modelling and design in Ptolemy II. Figure 3-2 shows a simple example of a composite actor in Ptolemy II.
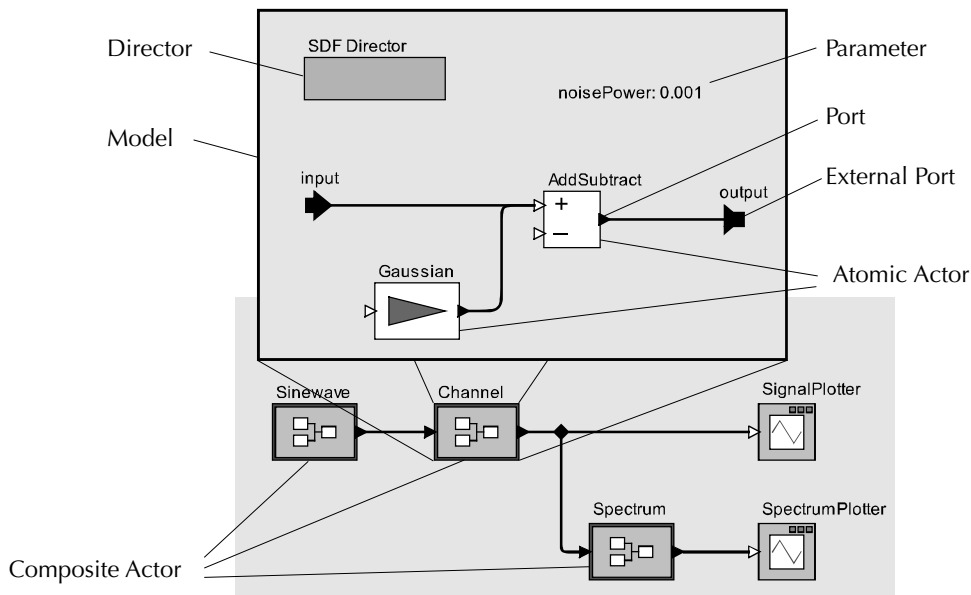
Fig. 3-2: A composite actor in Ptolemy II, which implements a simple communication channel with gaussian noise.

In actor models with composite actors, there has to be some coordination between the execution on the outside and the execution on the inside of actors. This coordination of possibly different models of computation is one of the main research areas of the Ptolemy project and is defined by the *abstract semantics* of a system.

In Ptolemy II, the *Executable* and *Receiver* interfaces abstract the flow of control and the communication semantics respectively. These two interfaces together, define a suite of methods, the semantics of which are the abstract semantics of Ptolemy II.

3.4 Abstract Semantics

The abstract semantics of Ptolemy II abstracts how communication and flow of control work, and is what makes it possible to compose models hierarchically and heterogeneously [14]. It does this, by defining an intersection of interesting semantics to represent common features of all models of computations.

Actors that obey the abstract semantics of Ptolemy II, can usually be used in models with any MoC that also conforms to this abstract semantics. In Ptolemy II, such actors are called *domain polymorphic* actors.

3.4.1 Abstract Flow of Control

In the Ptolemy II abstract semantics, actors execute in three phases:

- *initialization*

- a sequence of *iterations*

- *wrap-up*.

An iteration is a sequence of operations that read input data, produce output data, and update the state. An iteration of an actor is also often called its *firing*. In the abstract syntax of Ptolemy II, the firing of an actor is again divided into three separate phases, which is sometimes referred to as split phase firing. During one iteration, the director which controls the actor makes:

- exactly one invocation of *prefire*,

- any number of invocations of *fire*,

- at most one invocation of *postfire*.

In the prefire method, an actor may determine, whether its conditions for firing are satisfied or not. A typical condition for firing is for example the availability of tokens on the input ports of the actor.

If the actor indicates to the director, that its preconditions are satisfied, the iteration proceeds with one or more executions of fire, followed by exactly one invocation of postfire. The fire method is the main point of execution and is generally responsible for reading inputs and producing outputs, but it must not update the state of the actor. Instead, updating the state of the actor is the responsibility of the postfire method, which is invoked at the end of an iteration.

Interestingly to note is, that only the fire and postfire methods may consume tokens, and only the fire method is supposed to produce tokens.

3.4.2 Abstract Communication

The abstract semantics of Ptolemy II provides a set of primitive communication operations, which allow an actor to query the state of communication channels and to retrieve or send information from and to channels:

- *get* retrieves a data token via a port,

- *put* produces a data token via a port,

- *hasToken(k)* tests whether *get* can be successfully applied *k* times to a port,

- *hasRoom(k)* tests whether *put* can be successfully applied *k* times to a port.

These communication operations might be implemented differently, depending on the MoC and the "mechanics" of communication defined by it. For example in a SDF MoC, the channel is implemented as a fixed-sized FIFO buffer, while in a CT MoC, the channel is a simple variable whose value is the value of a signal at the current time. By only using the abstract communication operations, an actor is not affected by how the underlying communication channel is implemented and is therefore domain-polymorphic from the communications point of view.

Chapter 4

CAL – The Cal Actor Language

CAL is a textual language for writing actors, which was created as a part of the Ptolemy II project at UC Berkeley. This chapter gives a short introduction into the CAL language. More information on CAL can be found in [10], [9] and on the Caltrop project web-page [2].

4.1 Purpose and Goals of CAL

CAL is a domain specific language for defining the functionality and behaviour of actors. Its key goal is to make actor programming easier, by providing a concise high-level description of an actor. This enables an actor programmer to express the information and behavioural properties of an actor that are relevant to its usage, but that would be only implicit in a description of the actor in a traditional programming language such as C or Java.

Besides the above-mentioned goal, CAL is also designed to facilitate the automatic extraction of actor properties needed, for example, to analyze actors and actor compositions.

CAL is not intended to be a full-fledged programming language, but to be embedded in a richer environment which provides the necessary infrastructure. The language does not specify a strict semantics for all the constraints of an actor, such as the type system, communication mechanisms or scheduling schemes, nor does it define a syntax to define composite actors.

4.2 A Simple Example of a CAL Actor

To get an impression of CAL let's look at Example 4-1, which shows a *Ramp* actor written in CAL. The *Ramp* actor creates a sequence of tokens, where the first token's value equals the parameter `init` and the value of each following token is increased by the parameter `step`. Each time the *Ramp* actor gets a trigger-token on its input, it creates a new token on its output.

Example 4-1:

```
1:  actor Ramp (Integer init=0, Integer step=1) Integer In ==> Integer Out:
2:      Integer state := init;
3:      Action_1:action In:[trigger] ==> Out:[out]
4:          var Integer out = state;
5:          do
6:              state := state + step;
7:          end
8:      end
9:  end
```

Actor Header

The first line of the Ramp actor is the actor header and defines the interface of the actor to its environment. The actor header contains the actors name, parameters and port definitions. Additionally the actor header could also specify a number of type constraints, which is not shown here.

Let's take a closer look to the actor header of the Ramp actor:

- The **actor** keyword starts the *actor definition* and is followed by the actor name `Ramp`.

- Following the actors name, a pair of parentheses enclose the *parameter declarations* of the actor. Each parameter declaration consists of a type, the parameter name and a default value. The *Ramp* actor declares two parameters, namely `init` and `step`.

- After the parameter declarations the last part of the action header is formed by the *port declarations*. The arrow separates the input port declarations on its left side from the output port declarations on its right side. Every port declaration consists of a port type and a port name, where the port type declares the type of the tokens arriving or leaving on the port. The *Ramp* actor declares one input port `In` and one output port `Out`.

- A colon marks the end of the actor header and the beginning of the *actor body*.

State Variable Declaration

Following the actor header, on line two, the *state variable declarations* of the actor are defined. The values of the state variables are persistent between firings of the actor. The *Ramp* actor has one state variable named `state`.

Action Definition

Starting on line three, the action of the *Ramp* actor is defined. In general, a CAL actor may define any number of actions. Every action defines a separate action scope, which may contain any number of action variable declarations and statements.

Let's take a closer look to the action definition:

- The **action** keyword marks the beginning of the *action definition* and is preceded by an optional *action tag*, `Action_1` in our example.

- Following the action tag is the *port pattern* of the action. The arrow separates the action input patterns on its left side from the action output expressions on its right side.

- The *input patterns* specify how many tokens the action consumes from each input port and declare action variables to which the consumed tokens are bound to. Optionally an input pattern may be preceded by a *port tag* to identify the port it belongs to. In our example the action consumes one token from the input port `In` and binds this token to a variable with name `trigger`.

- The *output expressions* specify how many tokens the action produces and what values the produced output tokens have. Like input patterns, an output expression may optionally be preceded by a port tag. In our example the action produces one token with the value `out` on the output port `Out`.

- Following the **var** keyword, an action may declare any number of *action variables* that are visible inside the action scope during one firing.

- Starting with a **do** keyword, the *action body* is defined. It may contain any number of statements that are executed each time the action is executed and that may change the actor state. The action body in our example has one statement on line six, which updates the state variable named `state`.

Everything up to the start of the action body is usually called the *header* of an action.

4.3 An Introduction to the Syntax and Semantics of CAL

4.3.1 Data Types and Data Structures

As mentioned earlier, CAL is almost totally agnostic with respect to the type system used, and the type system is considered to be a part of the host environment, in which CAL actors are embedded into.

Nevertheless, CAL defines a number of built-in types that are the types of objects created as the result of special language constructions and it also defines a number of data structures like sets, lists and maps.

For our use of CAL, the type system is irrelevant and is therefore not discussed in more detail.

4.3.2 Expressions and Statements

Expressions in CAL are free of side-effects and strongly typed. CAL provides several individual kinds of expressions, some of which are:

- *Literals*, such as 1, 1.414 or 'hello world!'

- *Identifiers*, such as variable names, port names, etc.

- *List, Set, Map* comprehensions

- *If-then-else expressions*

- *Closures*, such as *Lambda expressions* or *Proc expressions*

Statements on the other side and may change the state of an actor. They are executed sequentially, in the order in which they appear in the code. CAL provides several individual kinds of statements, some of which are:

- *Assignments*, such as `init=1`

- *Flow-Control-statements*, such as *If-then-else*, *While* or *Foreach*.

- *Call-statements*

4.3.3 The Structure of a CAL Actor

Example 4-2 shows the skeleton of a CAL actor and introduces the names for the different grammatical elements of an actor. These are described in the following chapters.

Example 4-2:

```
1:  actor actor_name (parameter_declarations)
            input_port_declarations ==> output_port_declarations:
2:      ...
3:      state_variable_declarations
4:      ...
5:      initalization_statements
6:      ...
7:      action_tag:action input_patterns ==> output_expressions
8:          guard guard_conditions
9:          var action_variable_declarations
10:         do
11:             action_body
12:         end
13:     end
14:     ...
15:     action
16:         ...
17:     end
18:     ...
19:     selector
20:         ...
21:     end
22:end
```

4.3.4 Parameter Declarations

The parameter declarations define a number of parameters which can be set by the host environment when initializing an actor. After initialization, the parameters are constants within the actor scope. CAL allows to define a default value for every parameter.

4.3.5 Port Declarations

The port declarations define the number of input and output ports as well as their names and types.

The type of a port specifies the type of the tokens consumed or produced on a port and therefore also the type of the action variables that the tokens of a certain port are bound to.

4.3.6 State Variables and Initialization

The state of an actor is stored between firings. The code associated with an action can therefore use the state to influence the actions taken by future firings.

An actor may contain any number of state variable declarations, as well as a number of initialization statements that are executed when the actor is initialized.

4.3.7 Actions

Actions define atomic pieces of computation, performed by an actor, usually in response to some input from the environment. The definition of an action needs to describe three things:

- the *consumption* of input tokens,

- the *production* of output tokens,

- the *change of state* of the actor.

In CAL the consumption of input tokens is specified with an actors input patterns, the production of output tokens is specified with its output expressions and the change of state is defined in its action body.

Input Patterns

Firing an action may consume some tokens form the input ports of the actor and produce tokens on its output ports. On the input side, input patterns are used to describe the token consumption of a given action. An input pattern is said to *match* if enough tokens are available on its input ports.

Besides defining the number of tokens that are consumed by an action from a certain input port, the input patterns also introduce variables to which the values of the consumed tokens are bound to.

CAL defines several formats for patterns, of which only one is introduced here:

- $[t_1, ..., t_n]$ This pattern introduces n variables, which represent the first n tokens available on the channel. The first token is bound to the variable t_1, the second to t_2, and so on.

Input patterns may contain a so called *repeat expression*, which allows to dynamically determine the number of tokens that are matched by a pattern.

Output Expressions

On the output side of an actor, the output expressions are used to compute the tokens to be sent to the output ports. The general format of an output expression is very similar to that of an input pattern, except that instead of a channel pattern, the output expression contains a so called token expression that evaluates the tokens that are to be sent.

Same as input patterns, output expressions may also apply on single channels or multi channels and may contain repetition count.

Action Body

The action body is defined within a `do-end` construct and contains any number of statements that are executed when an action is selected for firing. The statements in the action body may change the actor state.

4.3.8 Action Matching

A significant part of the expressiveness of CAL comes form the way actions are chosen for firing. An actor may consist of any number of action definitions of which it has to select one when fired. The selected action is executed to consume input tokens, generate output tokens and to compute a new state for the actor.

An actor can only select an action that is said to be fireable. The parts of an action definition that are considered to determine an actors fireability are the following:

- The action selector.

- The input patterns.

- The local variable declarations in the `var`-clause.

- The boolean expressions in the `guard`-clause.

The action selector, which is explained in more detail below, pre-selects a set of actions that are active in the current state of the actor. Of the pre-selected actions, every action is fireable that matches the input patterns and whose `guard`-clauses all evaluate to `true`. If more than one action evaluates to be fireable in the current state, CAL does not define which of the fireable actions is selected, instead it is up to the environment to define how an actor has to behave in this situation.

The variables declared by the input patterns and those in the `var`-clause are all declared in the same scope, namely the action scope, and may depend on each other in complex ways. The only constraint is that they may not depend on each other in a circular fashion. The variable dependencies are as follows:

- Variables in input patterns depend on all free variables of any repeat expression.

- Variables in `var`-clauses depend on all free variables in their defining expressions.

- `guard`-clauses depend on all free variables in their boolean expression.

4.3.9 Action Selector

In CAL it is possible to constrain the set of actions which an actor is allowed to fire depending of the history of fired actions. The **selector**-clause in CAL is used to define the active actions in every state and identifies the actions via their action tag.

Currently CAL allows two interchangeable formats to define action selectors, either an action selector may be written as a regular expression or as a finite state machine.

Example 4-3: When the following actor is fired for the first time, only action s1 is active and may be selected to be fired. For any following firing however, only the actions s2 and s3 are active and may be selected to be fired.

```
1: actor A1 () T in1, T in2, T in3 ==> T out1, T out2:
2:     s1:action [a], [], [] ==> [d], [e] ... end
3:     s2:action [], [b], [] ==> [], [e] ... end
4:     s3:action [], [], [c] ==> [], [e] ... end
4:     selector
5:         s1(s2|s3)*;
8:     end
9: end
```
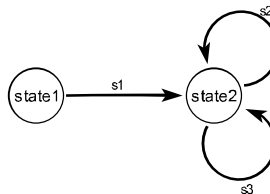

Fig. 4-4:   The finite state machine format of the **selector**-clause of the above example.

Calflow

Calflow is an intermediate format for actors written in Cal, and was created as part of this thesis. This chapter gives an introduction into Calflow. More information regarding Calflow can be found in Appendix A.

5.1 Purpose and Goals of Calflow

Much of Cal's expressiveness is based on its concept of action matching. While the body of an action in Cal is defined in an imperative way, the action header, which contains all the information needed for action matching, is rather declarative, and variables declared and used during action matching may depend on each other in complex ways.

In Calflow, these complex dependencies are explicitly resolved and actions are represented as dataflow graphs with states that represent atomic execution steps of actions.

Calflow seems to be an interesting intermediate format of Cal for a number of applications, most notably probably for code generation, or more general for the execution refinement of Cal. Calflow proves also to be useful for a number of actor analysis problems.

5.2 An Introduction to the Syntax and Semantics of Calflow

The actions of a Calflow actor are usually represented as a set of data flow graphs, with one graph for every action.

5.2.1 The Structure of a Calflow Actor

Example 5-1 shows the skeleton of a Calflow actor and introduces the names for the different grammatical elements of an actor. These are described in the following chapters.

Example 5-1:

```
 1: actor actor_name (parameter_declarations)
            input_port_declarations ==> output_port_declarations:
 2:     ...
 3:     state_variable_declarations
 4:     ...
 5:     initalization_statements
 6:     ...
 7:     action_tag:action
 8:         step_tag:atomicstep step_type
 9:             atomicstep_body
10:         end
11:         ...
12:         dependency dependency_type
13:             dependency_declarations
14:         end
15:         ...
16:         schedule
17:             action_schedule
18:         end
19:     end
20:     ...
21:     action
22:         ...
23:     end
24:     ...
25:     selector
26:         ...
27:     end
28:end
```

5.2.2 Relations between Calflow and CAL

In Calflow, all parts of an actor definition except the definition of its actions, are exactly the same as in CAL. Calflow also uses the same data types and data structures as well as the same expressions and statements. The following chapter explains how actions are defined in Calflow.

5.2.3 Actions

In Calflow, every action is defined as a dataflow graph, which consists of a set of *atomic steps*, the states of the graph, and a set of transitions, the *dependencies* between the atomic steps.

Optionally an action may also contain a schedule, called *action schedule*, which defines a partial or total order on all or some *atomic steps* of an action.

Atomic Steps

As mentioned above, *atomic steps* are the states of the dataflow graph that defines an action in Calflow. We think of them as atomic execution steps of an action that, when executed in a valid order, make up the complete execution of an action.

Each *atomic step* defines its own variable context for execution and may additionally have the following interactions with the action context and the actor context:

- it may *declare* any number of variables in the action context,

- it may *depend* on any number of variables in the actor context and

- it may *assign* a new value to any assignable variable in actor context.

Calflow defines seven different types of atomic steps:

- In an *InputGuard* atomic step, an action checks whether the input pattern of an input port matches the current availability of tokens on its channels. An action must have one *InputGuard* atomic step for every *Input* atomic step.

- During an *Input* atomic step, an action reads the tokens of one input port as specified by the input pattern of this port in CAL. An action must have one *Input* atomic step for every input port whose input pattern in CAL is not empty[1].

- In a *Decl* atomic step, an action declares exactly one variable in the action context. An action must have one *Decl* atomic step for every variable that is declared in its `var`-clauses in CAL.

- In a *Guard* atomic step, an action evaluates exactly one boolean expression. An action must have one *Guard* atomic step for every `guard`-clause it has in CAL.

- In an *OutputGuard* atomic step, an action checks whether the channels of an output port are ready to receive the number of tokens that are defined by the output expression of an output port in CAL. An action must have one *OutputGuard* atomic step for every *Output* atomic step.

---

[1] An action may or may not have (empty) Input atomic steps for input ports whose input patterns in CAL are empty.

- During an *Output* atomic step, an action produces the tokens on one output port as specified by the output expression of this port in CAL. An action must have one *Output* atomic step for every output port whose output expression in CAL is not empty[2].

- During a *Stmt* atomic step, an action executes exactly one statement of the action body in CAL. An action must have one *Stmt* atomic step for every statement that is contained in its action body in CAL.

Six of the seven types of *atomic steps* defined above can be categorized nicely by their functionality inside an action execution:

- the *InputGuard, OutputGuard* and *Guard atomic steps* determine the *fireability* of an action,

- the *Input atomic steps* handle the *consumption* of input tokens,

- the *Output atomic steps* handle the *production* of output tokens and

- the *Stmt atomic steps* handle the *change of state* of the actor.

Of a more general nature are *Decl atomic steps*, which may be needed to fulfill any of the above mentioned functionalities.

Dependencies

Every action must define a set of *dependencies* between the atomic steps of the action. The transitions in an action's dataflow graph represent these dependencies.

In the syntax of Calflow, each dependency may be categorized into either being a *data dependency* or a *constraint dependency*. Even though this categorization may be of interest for some analysis, the semantics of all dependencies is the same, namely that one atomic step depends in some way on the execution of the other atomic step.

As the name suggests, data dependencies exist between any two atomic steps of which the second one depends in any way on data that is either declared or changed by the first one. Additionally we introduce a data dependency between every InputGuard atomic step and the Input atomic step of the same input port, as well as between every OutputGuard atomic step and the Output atomic step of the same output port.

---

[2] As with the Input atomic steps, an action may or may not have (empty) Output atomic steps for output ports whose output expressions in CAL are empty.

Constraint dependencies on the other hand are dependencies that are not necessary from a dataflow point of view, but that are introduced by the environment to obtain a desired behavior of an action execution. In CAL, an action can only be executed when it is fireable, therefore any atomic step which is not side-effect-free to the environment, namely Output and Stmt atomic steps[3], should depend on all atomic steps that are needed to determine the fireability of an action, namely InputGuard, OutputGuard and Guard atomic steps.

Action Schedule

Every action of an actor may optionally contain an *action schedule* which defines a partial or total order on all or some atomic steps of the action. To be a valid action schedule the defined order must hold the dependency-relations of all atomic steps.

5.2.4 Action Matching

Action matching in Calflow works in the same way as in CAL. The action-selector preselects a set of actions that are active in the current state of the actor. Of the preselected actions, every action is evaluated on its fireability and if more than one action evaluates to be fireable in the current state, Calflow does not define which of the fireable actions is selected.

The difference between Calflow and CAL however is, that in Calflow we can see very explicitly the amount of execution needed to evaluate an actions fireability. Every action must execute all its *InputGuard*, *OutputGuard* and *Guard atomic steps* as well as any atomic steps they depend on to evaluate its fireability.

---

[3] It can be argued, that the Input atomic step, which consumes tokens, is also not side-effect-free to the environment. But the CAL semantics allows the action matching to be dependent on consumed tokens and therefore Calflow should not introduce a constraint dependency between Input atomic steps and the atomic steps that determine the fireability of an action. Furthermore it is often possible to work around the illegal consumption of tokens before action matching, for example by pushing the token back or by internally buffering it. However, these work-arounds are MoC dependent.

5.3 A Simple Example of a Calflow Actor

In this chapter we will use the again the Ramp actor which we already introduced previously. Example 5-2 shows the Calflow code of the Ramp actor. The CAL code of the same actor can be found in Example 4-1.

Example 5-2:

```
1: actor Ramp (Integer init=0, Integer step=1) Integer In ==> Integer Out:
2:     Integer state := init;
3:     A_1:action
4:         s1:atomicstep InputGuard
5:             -> s2;
6:         end
7:         s2:atomicstep Input
8:             In:[trigger];
9:         end
10:        s3:atomicstep OutputGuard
11:            -> s4;
12:        end
13:        s4:atomicstep Output
14:            Out:[out];
15:        end
16:        s5:atomicstep Decl
17:            out = state;
18:        end
19:        s6:atomicstep Stmt
20:            state = state + step;
21:        end
22:        dependency Constraint
23:            s1 -> s4;
24:            s1 -> s6;
25:            s3 -> s4;
26:            s3 -> s6;
27:        end
28:        dependency Data
29:            s1 -> s2;
30:            s3 -> s4;
31:            s5 -> s4;
32:        end
33:    end
34:end
```

Figure 5-1 shows the dataflow graph representation of the action in the Ramp actor and Figure 5-2 shows the action with a total order on its action steps. Details on the scheduling algorithm used to generate the action schedule in Figure 5-2 can be found in Appendix A.
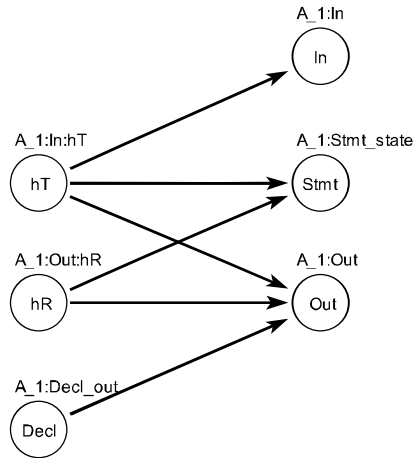
Fig. 5-1: The dataflow graph representation of the action in the Ramp actor. The states are labelled according to the atomic step they represent: InputGuard →hT, Input →In, Decl →Decl, OutputGuard →hR, Output →Out, Stmt →Stmt. The labelling of the InputGuard and the OutputGuard is modelled after the Ptolemy II methods hasToken() and hasRoom(). The name of every state is assembled from the action name, the port name if applicable and an identifying string.

In this particular example, hT corresponds to the atomic step s1, hR to s3, Decl to s5, In to s2, Stmt to s6 and Out to s4.
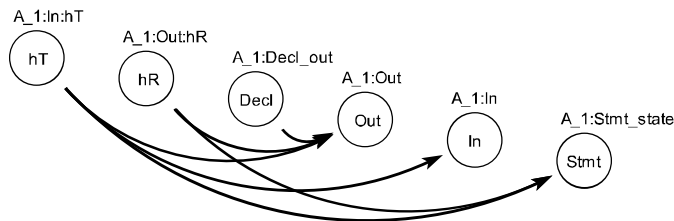
Fig. 5-2: The atomic steps of the action in the Ramp actor with a total order, defined by an action schedule.

Part II

Solution

Chapter 6

Overview

This chapter gives an overview of the strategy for static analysis of actor systems, which we present in this thesis.

Instead of analyzing actor systems directly, we use a light-weight formalism, based on Interface Automata [5], to capture certain aspects of the actors and the system environment, which are important for our analysis.

For this purpose, we introduce a new kind of interface automata, which we call Counting Interface Automata (CIA). Counting Interface Automata are based on Interface Automata and provide the capability of counting with natural numbers.

Using CIA, we can capture both, the interface description of an actor as well as parts of its component description [6], in particular its token consumption and production rates. Additionally, we strenghten its interface description by enabling it to express not only temporal, but also quantitative aspects.

Figure 6-1 shows the relation of an actor model and its composition on the top and the related interface automata on the bottom.

The transformation c_{MoC}, is a program transformation from an actor system with a particular model of computation into its composition [12]. This transformation leads to a complete description of the composite actor system.

For the analysis of the actor system however, we suggest to build the composite actor system in the interface automata domain. There, the transformation c_{MoC} becomes a transformation $\bar{c}_{MoC}$ from an actor system framework to its interface automaton. Since the interface automaton only captures certain aspects of the system, it is in general easier to find $\bar{c}_{MoC}$ than c_{MoC}. And additionally, the transformation $\bar{c}_{MoC}$ will probably be less expensive.
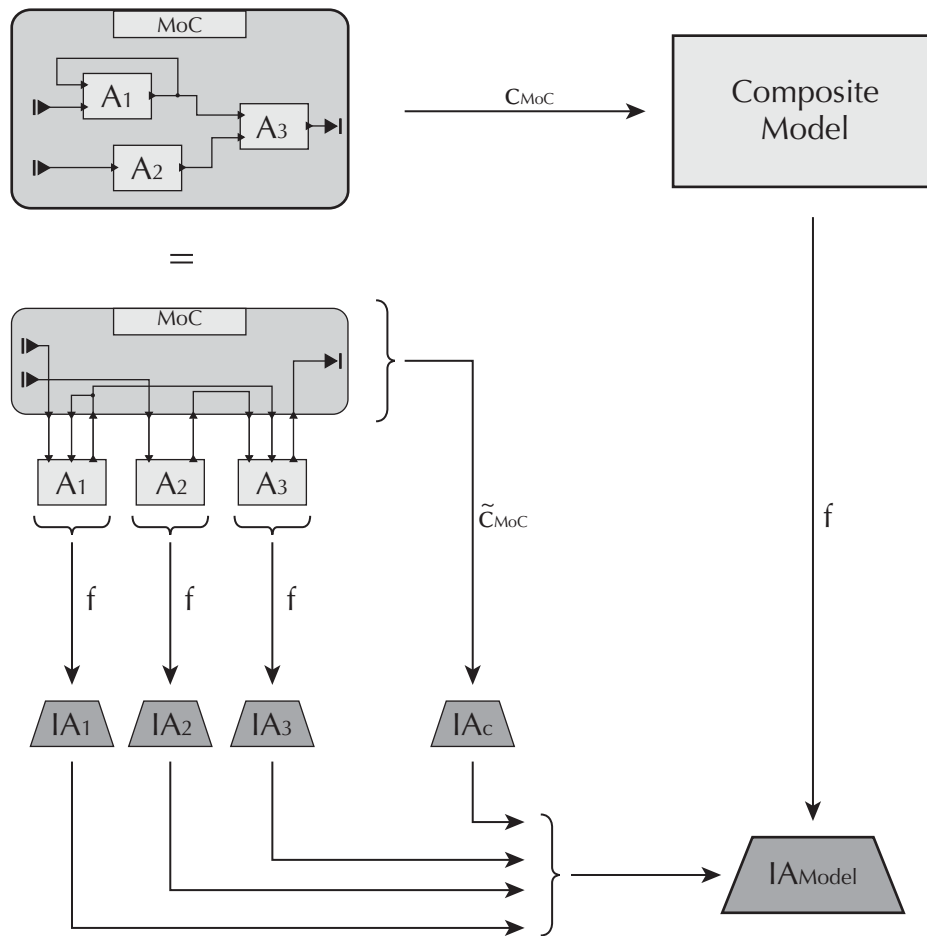
Fig. 6-1:   The relation of an actor system, its composition and interface automata.

The presented analysis strategy leads to a process, which can be broken down into three separate steps:

- *generation* of interface automata for the actors and the system,

- *composition* of the generated automata,

- *analysis* of the resulting composition automaton.

During the first step, we extract certain aspects of the interface description as well as the component description of both, the actors and the environment in which the actors are embedded in, and describe these extracted aspects using interface automata.

Then, in the second step, the successful composition of all obtained interface automata ensures compatibility of all component interfaces and results in an automa-

ton which captures the interface description of the composite actor as well as aspects of its internal component description.

In the third step, we then use the internal component description of the resulting automaton, to analyze a number of properties of the composite actor system, such as boundedness, deadlock or existence of legal firing sequences.

In the next chapter, we first describe and define Counting Interface Automata. The following three chapters then describe the three steps of the presented process.

Chapter 7

Counting Interface Automata

7.1 Preview

Counting Interface Automata (CIA) are based on Interface Automata [5] and provide the capability of counting with natural numbers. As other interface automata, CIAs consist of states and transitions and are usually depicted by bubble-and-arc diagrams.

There are three different kinds of transitions in a CIA: input, output and internal transitions. When modeling a software component, input transitions correspond to the invocation of methods on the component, or the returning of method calls from other components. Output transitions correspond to the invocation of methods on other components, or the returning of method calls from the modeled component. Internal transitions correspond to computations inside the component.

All transitions are labeled by actions, which are also divided into input, output and internal actions, depending on the kind of transition they label.

As the name suggests, Counting Interface Automata have the ability to count with natural numbers. To keep track of counted values, a CIA uses a set of so called counter variables. The natural numbers in CIA may represent anything countable of interest, as for example data tokens or discrete time. In these two cases, counter variables could represent the length of token queues or the current time. Sometimes we want also to express a value, which might not be known at construction time, but which is known to be constant. For these cases, CIA allows counter variables to be symbolic constants.

Every transition may contain a guard, which expresses a condition on the counter variable values, and which determines, whether the transition is enabled and can be taken or not. Furthermore, a transition may contain a set of counter declarations and counter assignments, which declare new counter variables or update their values respectively when the transition is taken.

So-called quantities on input and output transitions are used to express quantitative aspects of actions in a transition and may also be used to exchange counter values between two CIAs during composition. When modeling a software component, a

quantity could correspond to a countable argument of a method call, as for example a number of data tokens that should be passed with a method call or an amount of time that elapses during a method call. Quantities are particularly interesting during composition, where the equality of quantities is an additional constraint on synchronized shared transitions. Usually, a quantity of a transition is a natural number or the value of a counter variable.

There are however two special cases, namely so-called input declaration quantities and symbolic output declaration quantities. In a symbolic output declaration quantity, the quantity becomes the value of a symbolic constant, which gets additionally bound to a counter variable of the automaton. Input declaration quantities on the other side are open and may take on any value during composition with another automaton, this value then gets bound to a counter variable. In particular, on shared synchronized transitions, where the quantity of the transition of one automaton is an input declaration quantity, the value of the quantity of the output transition of the other automaton is bound to the counter variable of the input declaration quantity. This mechanism allows to exchange counter values between two automata during composition.

A transition in a CIA may consist of a number of sub-transitions. All sub-transitions of a transition must start at the same state but may end at different states. Every sub-transition contains a predicate which determines whether the sub-transition can be taken or not. If more than one predicate is true, then the choice among these sub-transitions is non-deterministic.

The composition of two CIA is only defined, if their counter variables and the sets of input actions and output actions respectively are disjoint. However, an input action of one CIA may coincide with an output action of the other CIA. The two automata will synchronize on such shared actions if either the quantities of both actions are equal or the quantity of the input action is an input declaration quantity, and they will asynchronously interleave all other actions.

7.1.1 A Simple Example of Counting Interface Automata

Figure 7-1 shows the CIA of a simple DDF director, which requires from its actors to check for the availability of tokens prior to consuming any.

The director has a counter variable T, which represents the number of tokens that are available in the buffer of the receiver.

State 0 is the initial state of the director. Before being ready to receive any input, the director needs to execute some internal operations, which is expressed by a transition with an internal action *step*. In state 1, the director is ready to receive a method
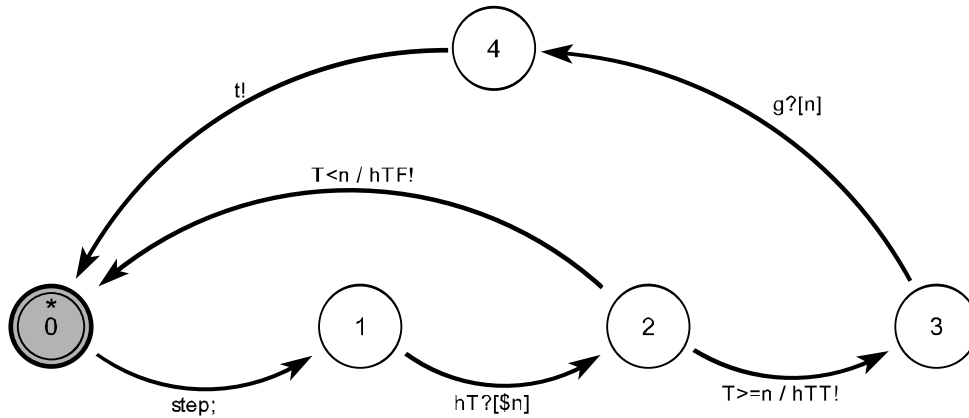
Fig. 7-1: The CIA of a simple DDF director. Input, output and internal actions are labelled with "?", "!" and ";" respectively. Note that quantities equal 1 are usually not shown explicitly.

call *hasToken(n)* from an actor, which is expressed by the input action *hT*. The transition has an input declaration quantity, which means that the calling actor can pass an argument, namely its own quantity, which is then bound to the counter variable *n*. If enough tokens are available in the receiver, the director returns true, which is expressed by the output action *hTT*. Otherwise it returns false, which is expressed by the output action *hTF*. If the director returned true, the actor may get exactly *n* tokens from the receiver. This is expressed with the input action *g*, which has a fixed quantity, namely the value of *n*. Finally, the director returns the tokens and goes back into its initial state.

7.2 Definition

Counting interface automata provide an interface automaton with the ability to count and to express quantitative aspects on transitions. In particular, counting interface automata use a set C of *counter variables* to track counted values. The values of these variables are elements of $N_0 \cup K \cup \{\perp\}$, where N_0 is the set of all natural numbers including zero, K is an infinite set of symbolic constants and $\perp$ represents an uninitialized counter variable.

Before we can define counting interface automata, we need to define some related elements.

Definition 1: (counter valuation) A *counter valuation V* over a set *C* of counter variables is a function $V: C \to N_0 \cup K \cup \{\bot\}$.

Definition 2: (counter operations) We define a set of different counter operations and functions.

- A *counter assignment CA* is a counter operation defined as
  $CA = \{x \lozenge y: x, y \in C\} \cup \{x \lozenge n: x \in C, n \in (N_0 \cup K)\}$, where $\lozenge$ is either the *increment*, *decrement* or *set* operation.
  We further define a function *Asgn*: $2^{CA} \to (V \to V)$, which maps a counter valuation to a new counter valuation, by applying a set of counter assignements.

- A *counter declaration CD* is a counter operation defined as
  $CD = \{x:=y: x, y \in C\} \cup \{x:=n: x \in C, n \in (N_0 \cup K \cup \{\bot\})\}$.
  We further define a function *Decl*: $2^{CD} \to (V \to V)$, which maps a counter valuation to a new counter valuation, by applying a set of counter declarations.

- An *input quantity counter declaration* CQ^I is a counter operation defined as
  $CQ^I = \{x:=n: x \in C, n \in (N_0 \cup K \cup \{\bot\})\}$ and an *output quantity counter declaration* CQ^O is defined as $CQ^O = \{x:=k: x \in C, k \in K\}$.
  We further define a function $Decl^Q$: $CQ \to (V \to V)$ with $CQ = CQ^I \cup CQ^O$, which maps a counter valuation to a new counter valuation, by applying a quantity counter declaration.

- A *counter guard CG* is a counter operation defined as $CG = \{x \lozenge y: x, y \in C\} \cup \{x \lozenge n: x \in C, n \in (N_0 \cup K)\} \cup \{CG \lozenge' CG\} \cup \{\neg CG\}$ where $\lozenge$ is either $<, \leq, =, \neq, \geq$ or $>$ and $\lozenge'$ is the boolean *AND* operator.
  We further define a function *Grd*: $CG \to (V \to \{true, false\})$, which maps a counter valuation into a boolean value, by applying a counter guard.

Definition 3: (choice predicates) The set of choice predicates *P* is defined as $P = 2^{P_C}$, where $P_C = \{p \in L_P\} \cup \{P_C \wedge P_C\} \cup \{\neg P_C\}$ and L_P is a set of boolean variables.

We are now ready to define counting interface automata.

7.2.1 Automaton

Definition 4: (counting interface automata) A *counting interface automaton* (CIA) is a

tuple $M = (C_M, S_M, s_M^{init}, A_M, Inv_M, T_M)$ consisting of the following components.

- C_M is a finite set of *counters.*

- S_M is a finite set of *states.*

- $s_M^{init} \in S_M$ is the *initial state.*

- $A_M = A_M^I \cup A_M^O \cup A_M^H$ is a finite set of *actions*, where A_M^I, A_M^O, A_M^H are mutually disjoint sets of *input, output* and *internal actions.*

- $Inv_M : S_M \rightarrow CG$ maps each state of M to its *state invariant.*

- $T_M = T_M^I \cup T_M^O \cup T_M^H$ is a finite set of *transitions*, where
$$T_M^I = S_M \times A_M^I \times Q_M^I \times CG_M \times 2^{CD_M} \times 2^{CA_M} \times 2^{P_M \times S_M}$$
$$T_M^O = S_M \times A_M^O \times Q_M^O \times CG_M \times 2^{CD_M} \times 2^{CA_M} \times 2^{P_M \times S_M}$$
$$T_M^H = S_M \times A_M^H \times \varnothing \times CG_M \times 2^{CD_M} \times 2^{CA_M} \times 2^{P_M \times S_M}$$
are mutually disjoint sets of *input, output* and *internal transitions* respectively.

  We can write each transition $t = (s, a, q, g, D, B, PS) \in T_M$ as a set of sub-transitions $\tau_i = (s, a, q, g, D, B, p_i, s_i') \in T_M$, $0 \leq i < |PS|$. For each of these sub-transitions, s and s_i' are the source and destination states of the sub-transition and $a \in A_M$ is an action, labeling the transition. $q \in Q_M$ is a quantity on the transition and is an element of a finite set of quantities $Q_M = Q_M^I \cup Q_M^O$, where $Q_M^I = N_0 \cup C_P \cup CQ^I$ and $Q_M^O = N_0 \cup K \cup C_P \cup CQ^O$ are sets of *input* and *output action quantities* respectively. Further, $g \in CG_M$ is a guard on the counter values that determines, whether a transition can be taken, $D \in 2^{CD}$ is a set of counter declarations and $B \in 2^{CA}$ a set of counter assignments that are applied when the transition is taken. Finally, $p_i \in P_M$ is a predicate that determines, whether the transition may choose s_i' as destination state.
  For any sub-transition relation, we may also write $s \xrightarrow{\tau_i} s_i'$.
  The guard $g \in CG_M$ of a transition is evaluated after applying any action quantity declarations, but before applying counter declarations or counter assignments.

Definition 5: (execution fragments) An *execution fragment* of a counting interface automaton M is an alternating sequence of states and sub-transitions $s_0, \tau_0, s_1, \tau_1, ..., s_n$ such that $s_i \xrightarrow{\tau_i} s_{i+1}$ for all $0 \leq i < n$.
An execution fragment is called *unconditional*, if it contains only output and internal

transitions and if for any transition, the conjunction of all choice predicates which remain if τ_i is removed, is not true. An execution fragment is called *conditional* otherwise.

The execution of a conditional execution fragment can be avoided by either the environment or the automaton, while the execution of an unconditional execution fragment cannot be avoided.

Definition 6: (reachability) Given two states $u, \ v \in S_M$, we say that v is reachable from u if there is an execution fragment whose first state is u, and whose last state is in v. The state v is reachable in M if it is reachable from the initial state s_M^{init} of M.

Definition 7: (execution paths) The *unconditional execution path set* $\Xi_M^{uc}(v)$ of a state v, is the set of all unconditional execution fragments through which v is reachable in M. The *conditional execution path set* $\Xi_M^{cond}(v)$ of a state v, is the set of all conditional execution fragments through which v is reachable in M.

Definition 8: (counter context) The *counter context Con* is a function $Con: \Xi_M(v) \rightarrow V(C_M)$, which maps an execution path of a state v to a valuation of C. It does so, by evaluating the counter valuation $V(C)$ along the execution path, starting at the initial state. For every sub-transition $s \overset{\tau_i}{\rightarrow} s'_i$ along the execution path, the counter valuation $V(C')$ in s'_i is obtained as follows.

$$V(C') = \begin{cases} Asgn(B, Decl(D, Decl^Q(q, V(C)))) & \text{if } q \in CQ \\ Asgn(B, Decl(D, V(C))) & \text{if } q \notin CQ \end{cases}$$

7.2.2 Product

Two CIAs M and N are *composable* if $A_M^I \cap A_N^I = \varnothing$, $A_M^O \cap A_N^O = \varnothing$ and $C_M \cap C_N = \varnothing$.
Their *shared actions* are $shared(M, N) = (A_M^I \cap A_N^O) \cup (A_M^O \cap A_N^I)$.

Definition 9: (product) For two composable CIAs M and N, the *product $M \otimes N$* is the CIA that consists of the following components.

- $C_{M \otimes N} = C_M \cup C_N$.

- $S_{M \otimes N} = S_M \times S_N$.

- $s_{M \otimes N}^{init} = s_M^{init} \times s_N^{init}$.

- $A^I_{M \otimes N} = A^I_M \cup A^I_N \setminus shared(M, N)$,
  $A^O_{M \otimes N} = A^O_M \cup A^O_N \setminus shared(M, N)$,
  $A^H_{M \otimes N} = A^H_M \cup A^H_N \cup shared(M, N)$.

- $Inv_{M \otimes N}(u, v) = Inv_M(u) \wedge Inv_N(v) \wedge Inv^T_{M \otimes N}(u, v)$, with

$$Inv^T_{M \otimes N}(u, v) = \begin{cases} Inv^{Ta}_{M \otimes N}(u, v) & \text{if } q^I \notin CQ^I \\ Inv^{Tb}_{M \otimes N}(u, v) & \text{if } q^I \in CQ^I \end{cases}$$

$Inv^{Ta}_{M \otimes N}(u, v) = \Lambda \left(\neg(\neg g^I \wedge g^O) \wedge \neg(g^I \wedge g^O \wedge \neg(q^I = q^O)) \right) \big|$
$\left(\left((u, a, q^I, g^I, D_M, B_M, PS_M) \in T^I_M \wedge (v, a, q^O, g^O, D_N, B_N, PS_N) \in T^O_N \right) \vee \right.$
$\left. \left((u, a, q^O, g^O, D_M, B_M, PS_M) \in T^O_M \wedge (v, a, q^I, g^I, D_N, B_N, PS_N) \in T^I_N \right) \right) \wedge$
$a \in shared(M, N)$

$Inv^{Tb}_{M \otimes N}(u, v) = \Lambda \left(\neg(\neg g^I \wedge g^O) \right) \big|$
$\left(\left((u, a, q^I, g^I, D_M, B_M, PS_M) \in T^I_M \wedge (v, a, q^O, g^O, D_N, B_N, PS_N) \in T^O_N \right) \vee \right.$
$\left. \left((u, a, q^O, g^O, D_M, B_M, PS_M) \in T^O_M \wedge (v, a, q^I, g^I, D_N, B_N, PS_N) \in T^I_N \right) \right) \wedge$
$a \in shared(M, N)$

- $T_{M \otimes N} = T^M_{M \otimes N} \cup T^N_{M \otimes N} \cup T^{MN}_{M \otimes N}$, with

$T^M_{M \otimes N} = \{ ((u, v), a, q, g, D, B, (p_i, (u_i', v))) \big|$
$(u, a, q, g, D, B, (p_i, u_i')) \in T_M \wedge a \notin shared(M, N) \wedge v \in S_N,$
$0 \leq i < |PS_M| \}$

$T^N_{M \otimes N} = \{ ((u, v), a, q, g, D, B, (p_i, (u, v_j'))) \big|$
$(v, a, q, g, D, B, (p_j, v_j')) \in T_N \wedge a \notin shared(M, N) \wedge u \in S_M,$
$0 \leq j < |PS_N| \}$

$T^{MN}_{M \otimes N} =$
$\{ ((u, v), a, \emptyset, g_M \wedge g_N, D_M \cup D_N, B_M \cup B_N, (p_{Mi} \wedge p_{Nj}, (u_i', v_j'))) \big|$
$(u, a, q_M, g_M, D_M, B_M, (p_{Mi}, u_i')) \in T_M \wedge$
$(v, a, q_N, g_N, D_N, B_N, (p_{Nj}, v_j')) \in T_N \wedge$
$a \in shared(M, N), \ 0 \leq i < |PS_M|, \ 0 \leq j < |PS_N| \}$

The invariants $Inv_{M \otimes N}(u, v)$ of the product $M \otimes N$ are the conjunction of the invariants of both automata M and N and additional transition invariants $Inv^T_{M \otimes N}(u, v)$. The transition invariants make sure, that for any shared transitions which are active in a state $(u, v) \in S_{M \otimes N}$, either the guard of the output transition is false or otherwise the guard of the input transition is true and the quantities match.

Definition 10:(error states) Given two composable CIAs M and N we define two types of error states.

- *Immediate error states*: A state $(u, v) \in S_{M \otimes N}$ is an immediate error state, if there is an action $a \in shared(M, N)$, such that there is an output sub-transition $u \xrightarrow{a} u'$ labeled with the action a for some state u', but no input sub-transition $v \xrightarrow{a} v'$ labeled with the action a for all states v', or such that there is an output sub-transition $v \xrightarrow{a} v'$ labeled with the action a for some state v', but no input sub-transition $u \xrightarrow{a} u'$ labeled with the action a for all states u'.

- *Counter error states*: A state $(u, v) \in S_{M \otimes N}$ is a counter error state, if for any unconditional execution path $\xi \in \Xi^{uc}_{M \otimes N}(u, v)$ or all conditional execution paths $\xi \in \Xi^{cond}_{M \otimes N}(u, v)$ the state invariant $Inv_{M \otimes N}(u, v)$ with the counter valuation $Con(\xi)$ evaluates to false.

7.2.3 Composition

Definition 11:(composition) The composition $M \,||\, N$ of two CIAs M and N is obtained by restricting the input and choice behavior of $M \otimes N$ to avoid all error states.

To restrict the input and choice behavior of $M \otimes N$, we remove all states in $S_{M \otimes N}$, from which there exists an unconditional execution fragment to any error state in $S_{M \otimes N}$.

Definition 12:(compatibility) Two CIAs M and N are compatible, if their composition is not empty.

Note, that in general, the composition and thus the compatibility of two counting interface automata is not decidable. But we will show in Chapter 9, that the CIAs we use have certain properties, which make composition and compatibility decidable.

7.3 Extension

We introduce an extension to Counting Interface Automata, which allows the formal treatment of structural properties that occur when using CIA in the context of actors and actor models as well as in the context of software components in general.

For this purpose, we first split the set of counters into two mutually disjoint sets of so-called *global counters* and *local counters*, and further, we define a subset of states, which we call *reset states*.

Together, the global and local counters as well as the reset states are important when evaluating the counter valuation along an execution path, in order to obtain the context of a state. For every sub-transition along an execution path, the counter valuation in the end-state of the sub-transition is defined in Definition 8. We extend this definition in a way, such that all local counter variables in the counter valuation of an end-state get uninitialized, if this end-state is a reset state.

With this extension, we distinguish two sets of counters, the values of one set are valid globally, while the values of the other set are only valid locally along the path between two reset states.

When we build the product of two CIAs, only the product of two reset states will again be a reset state. Thus, the product of a reset state with an ordinary state will be an ordinary state. However, the transition which leads to such an ordinary state must explicitly uninitialize all local counters which were previously uninitialized by the reset state that is not existing anymore.

Generating Counting Interface Automata

This chapter describes how counting interface automata of actors in general and models with dataflow MoC are generated.

8.1 Generating Actor Automata

In Figure 6-1, the transformation of actors from an actor language into interface automata is depicted as a function f. As mentioned earlier, we will use CAL as actor language and CIA as interface automata. Thus the function f becomes a transformation from CAL to CIA.
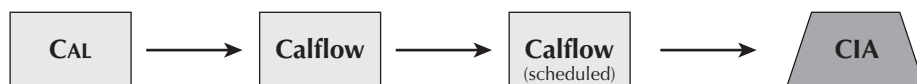
Fig. 8-1: Transformation steps to transform a CAL-actor into its CIA.

As shown in Figure 8-1, the transformation from a CAL-actor into its CIA can be broken down into three separate steps:

- transforming the CAL-actor into a Calflow-actor,

- adding an execution schedule to the Calflow-actor,

- transforming the scheduled Calflow-actor into its CIA.

The following chapters describe each of these three steps.

8.1.1 Transforming CAL into Calflow

The first step of transforming a CAL-actor into CIA, is to generate the previously described intermediate format Calflow from its CAL code. The exact transformation is described in Appendix A, since it is not of interest in this context.

8.1.2 Scheduling Calflow

After the transformation from CAL to Calflow, all actions in the Calflow actor will have an empty action schedule. However, to transform Calflow into CIA we need a sequential representation of CAL and therefore sequentially scheduled actions. For our purpose, any valid action schedule can be used. The used scheduling algorithm is described in detail in Appendix A.

8.1.3 Transforming Scheduled Calflow into CIA

The last step remaining, is to transform the scheduled Calflow-actor into its CIA. This process can be broken down into several steps, which are described below.

Creating the Director Interface
The actor CIA has two ports as interface to the director:

- *Director:f* is an input port, used by the director to fire the actor.

- *Director:fR* is an output port, used by the actor to return the fire command of the director.

Creating the Receiver Interfaces
For every input port of the actor, the actor CIA has five ports as interface to the related receiver:

- *Port:hT* is an output port, used by the actor to query the receiver whether it has a certain number of tokens in its buffer.

- *Port:hTT* is an input port, used by the receiver in return to a hT command if enough tokens are available in its buffer.

- *Port:hTF* is an input port, used by the receiver in return to a hT command if not enough tokens are available in its buffer.

- *Port:g* is an output port, used by the actor to get a certain number of tokens from the receiver.

- *Port:t* is an input port, used by the receiver in return to a get command.

For every output port of the actor, the actor CIA has also five ports as interface to the receiver of the connected actor:

- *Port:hR* is an output port, used by the actor to query the receiver whether it has enough room in its buffer for a certain number of tokens.

- *Port:hRT* is an input port, used by the receiver in return to a hR command if enough room is available in its buffer.

- *Port:hRF* is an input port, used by the receiver in return to a hR command if not enough room is available in its buffer.

- *Port:p* is an output port, used by the actor to put a certain number of tokens into the receiver.

- *Port:pR* is an input port, used by the receiver in return to a put command.

Creating Action Automata Sequences

To create the CIA of the actor, we first create CIA sequences for every action. Creating the CIA sequence of an action from its Calflow representation is a straightforward task, in which each atomic step is first mapped into a small sequence of a CIA. These small sequences are then assembled according to the Calflow action schedule. Finally, to build the complete CIA sequence of the action, a director fire input transition is added at its beginning and a fire return output transition at its end.

Figure 8-2 shows the mapping rules to map the atomic steps into CIA-sequences.

Creating the Actor Automaton

To create the CIA of the complete actor, we simply take the action selector of the Calflow actor in its finite state machine format and replace its states with CIA reset states and its transitions with the according CIA action sequences. If more than one transition leaves from the same state, we collect all director fire input transitions as sub-transitions into one transition with a true predicate on each sub-transition. This leads to a non-deterministic choice of all director fire input transitions.

If the Calflow actor has no action selector, this equals to an action selector with only one state and a cyclic transition for every action.
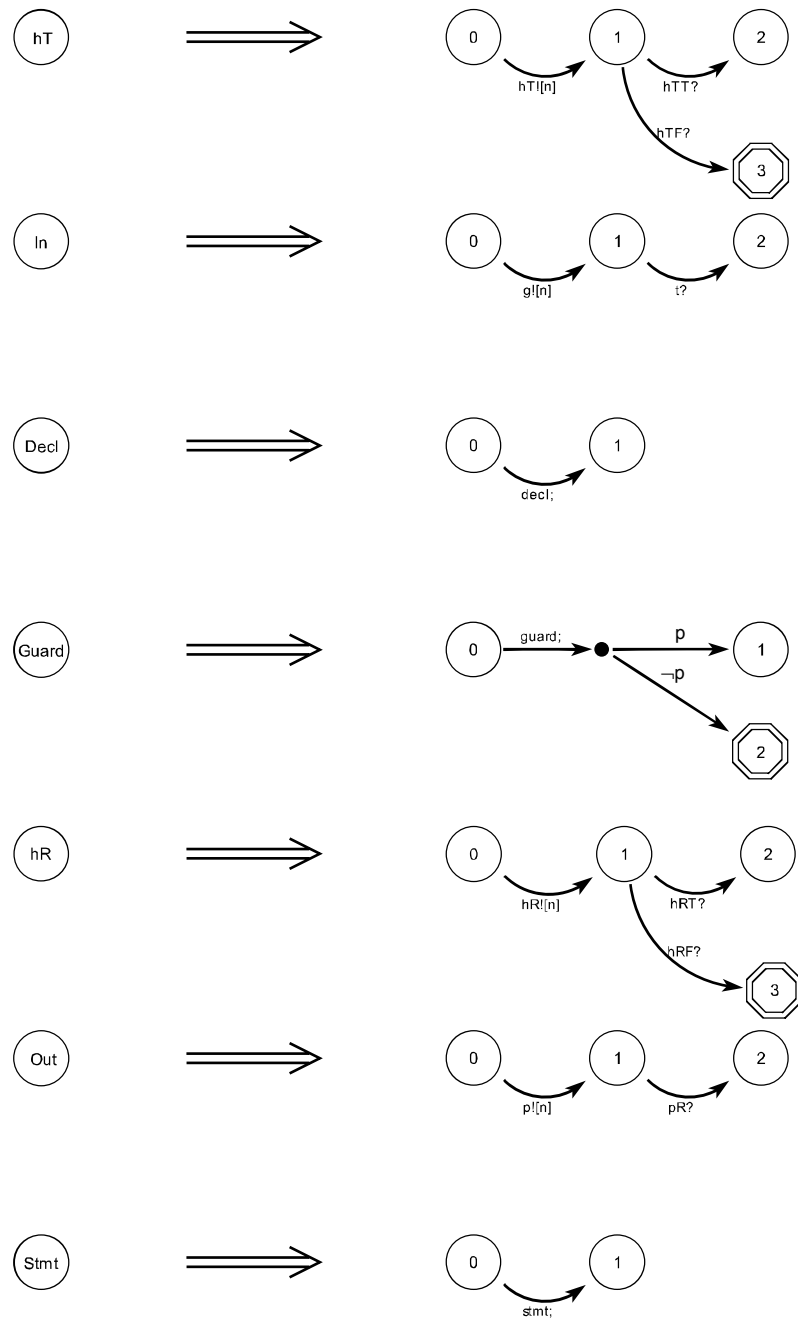
Fig. 8-2: Mapping rules to map atomic steps into sequences of CIA. The octagonal states are so called failure states. Their semantics is explained at the end of this chapter. The black dot in the guard mapping represents a choice on the transition.

8.1.4 A Simple Example of an Actor CIA

To understand how this works, let us analyse Example 8-1 and Example 8-2. They both show a Split actor written in CAL. On each firing, the Split actor consumes one token on its input port and forwards it to one of its two output ports. In Example 8-1, the order in which the actor selects the output port to forward the token to is determined by its action selector, while in Example 8-2, the order is non-deterministic.

Example 8-1:

```
1: actor Split () T In ==> T Out1, T Out2:
2:    A1:action [a] ==> [a],[] end
3:    A2:action [a] ==> [],[a] end
4:    selector
5:       ([A1][A2])*;
6:    end
7: end
```

Example 8-2:

```
1: actor Split () T In ==> T Out1, T Out2:
2:    A1:action [a] ==> [a],[] end
3:    A2:action [a] ==> [],[a] end
7: end
```

Figure 8-3 shows on its left side the unscheduled Calflow representation of the actions and on its right side the scheduled actions which are used to generate the CIA action sequences. Note, that no *OutputGuard* atomic steps are present in the Calflow actions, assuming that the environment provides unbounded buffers.

Figure 8-4 and Figure 8-5 show the complete actor CIA for the deterministic and the non-deterministic version of the Split actor respectively.
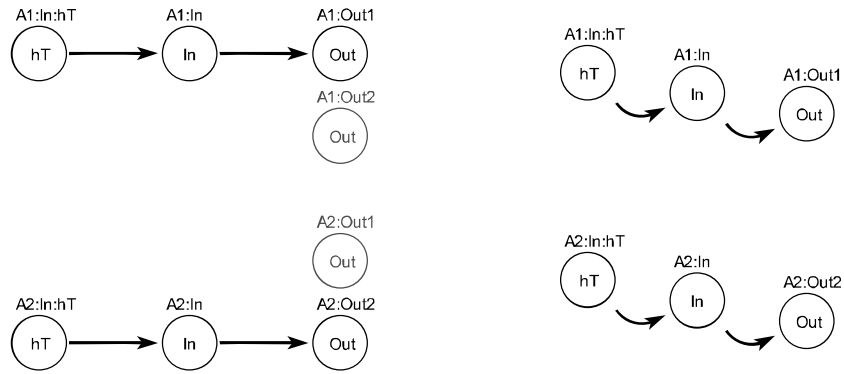
Fig. 8-3: On the left side are the unscheduled data-flow representations of the two actions of the Split actor. In both actions one of the output atomic steps is empty. On the right side are the scheduled actions. Note, the empty atomic steps are not scheduled by the scheduling algorithm.
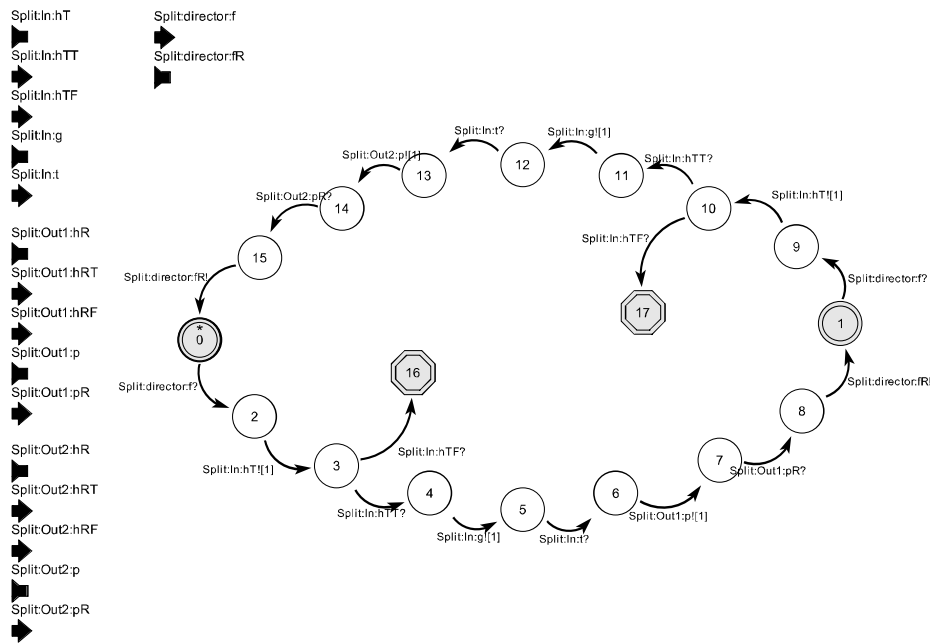
Fig. 8-4: The CIA of the deterministic Split actor.
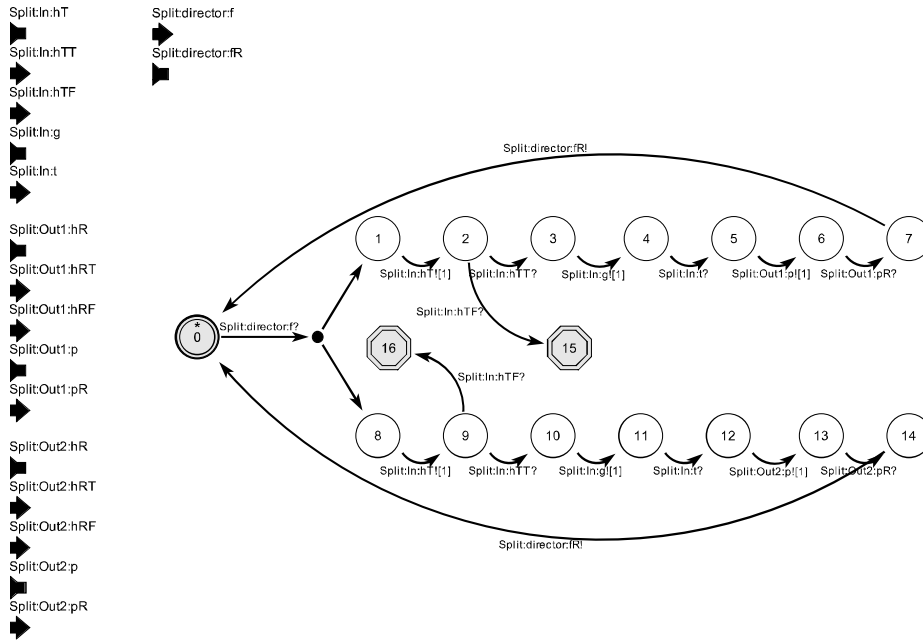
Fig. 8-5: The CIA of the non-deterministic Split actor.

8.2 Generating MoC Automata

In Figure 6-1, the transformation from the model connection information and the MoC into interface automata is depicted as a function $\bar{c}_{MoC}$. As mentioned earlier, we only generate various forms of dataflow MoC, the function $\bar{c}_{MoC}$ therefore becomes a function $\bar{c}_{xDF}$.

8.2.1 Dataflow MoC

In presented solution to create automata for dataflow models of computation leads to two design choices.

The first design choice is on whether the model of computation guarantees, that actions are only fired when enough tokens are available or not. If the model of computation gives this guarantee, then the actors are not required to check for tokens prior to consuming them.

The second design choice is on whether the buffers in our model are unbounded or not, and in the latter case whether the model of computation guarantees, that actions are only fired when enough room is available for all produced tokens. If the

buffers are bounded and the model of computation gives no guarantee on firing, then the actors are required to check for room in the connected receivers prior to producing tokens.

Additionally, a model of computation may have a partial or total order on the actor firing of the actors in a model.

In Ptolemy II, the SDF domain does not require its actors to check for tokens nor for room and it has a total order on the actor firing of all actors in a model.

On the other hand, for DDF models of computation, we will usually assume that the receivers have unbounded size and that the model of computation cannot give any guarantees on firing and has no fix order on the actor firing.

8.2.2 Creating Actor Automata Sequences

We start creating the CIA of a MoC by first separately creating CIA sequences to handle every actor in the model.

The basic structure of each of these actor CIA sequences is always the same, and starts with a CIA sequence to fire the actor, which leads to a state in which the actor is active and may execute. From there leaves again a CIA sequence to return the fire command.

We call the state in which the actor is active and may execute the *actor execution state*. In this state, the MoC must be ready to receive commands from the actor in any valid order. For this, a short transition sequence for every valid command leaves the actor execution state and returns again to it in a cycle. The valid execution order of the commands is then ensured with a set of temporary counters and guards on the command transitions.

The following sections describe in detail the steps needed to create an actor CIA sequence.

Creating Counters

One counter is created for every input port of the actor, to represent the number of tokens available in the receiver of this input port. This counter is usually named *Actor:Port:T*.

Optionally, if the receivers have finite buffer lengths, an additional counter is created for every input port of the actor, to represent the free space available in the receiver of this input port. This counter is usually named *Actor:Port:R*.

Creating the Director Interface

For every actor, the MoC CIA has two ports as interface to it. These ports are the same as for the actor CIA in Chapter 8.1.3, but with opposite directions.

Creating Receiver Interfaces

For every input or output port of every actor, which is not connected to an external port of the model, the MoC has five ports as interface to it. These ports are again the same as for the actor CIA in Chapter 8.1.3, but again with opposite directions. For ports that are connected to external ports of the model, the MoC does not provide an interface[1]. These ports will become ports of the composite actor.

Creating the Firing Automaton Sequence

The firing automaton sequence starts with an internal transition, labelled *step*, which signals the start of an action firing. This transition is followed by an output transition on the director fire port, which leads to the action execution state. From there an input transition on the fire return port ends the firing automaton sequence.

Creating Command Automata Sequences

For every input or output port of an actor a corresponding command automata sequence is added to the execution state of the actor. Figure 8-6 shows the different input and output automata sequences.

The left side of the figure shows three different input automata sequences. The uppermost is used if the MoC does not require the actor to check for tokens nor for room. The sequence in the middle is used if the MoC requires the actor to check for tokens but not for room. And finally the sequence at the bottom is used if the MoC requires the actor to check for tokens and for room.

The right side of the figure shows two different output automata sequences. The sequence at the top is used, if the MoC does not require the actor to check for room. The sequence at the bottom is used if the MoC requires the actor to check for room. The output sequence does not depend on whether the MoC requires the actor to check for tokens or not.

Note, how temporary counters and guards are used to ensure a valid execution order in cases where the MoC requires it. If the MoC for example requires the actor to check for tokens, a temporary counter, usually called *Actor:Port:cT* (cT is the abbreviation for "checked tokens"), is set with every has-token command of the actor. The

---

[1] Note that in order to automatically generate the CIA, every port of an actor in a model may be connected to either any number of ports of other actors in the model or to exactly one external port. We may however put identity-actors between ports and external ports, if a port needs to be connected to both an external port and ports of other actors.

Input Port Sequences   **Output Port Sequences**
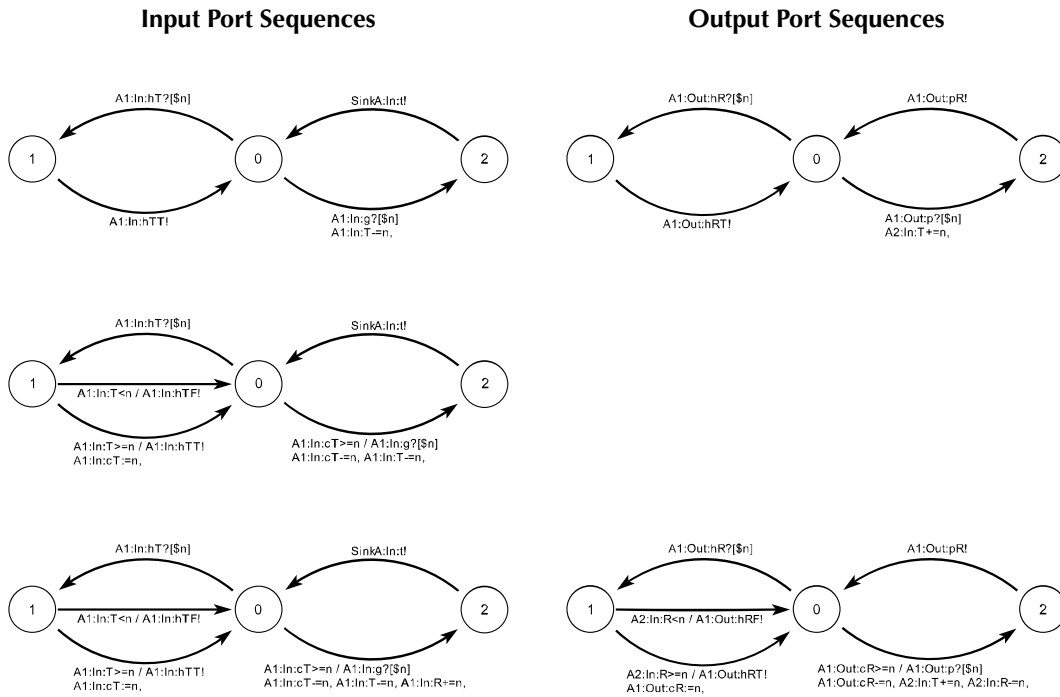
Fig. 8-6: Input and output automata sequences. The state 0 of every sequence represents the actor execution state. The sequences are shown for an actor named A1 with an input port In and an output port Out, which is connected to an input port In of an actor A2.

guard on the get command then ensures, that only as many tokens may be consumed as are set in this temporary counter. This ensures, that an actor cannot consume more tokens than he checked for. The actor may however consume the checked number of tokens in several separate get commands. The same technique is used if the MoC requires the actor to check for room.

8.2.3 Counters and Connection Information

The connection information of the original network is preserved in the counter assignments of the output automata sequences. During the put command on an output port, the transition assigns a new value to all counters that represent receivers to which the output port is connected to.

For example on the right side of Figure 8-6, the put command on the output port Out of actor A1 updates the token and room counters of the input port In of actor A2.

8.2.4 Creating the MoC Automaton

We assume that an actor execution schedule is available in form of a finite state machine, where the transitions represent actor firings. If no execution schedule is available, this equals to a finite state machine with only one state and a cyclic transition for every actor firing.

To create the complete MoC automaton, we take the actor execution schedule and replace its states with CIA rest states and its transitions with the according actor automata sequences. If more than one transition leaves from the same state, we collect all internal *step* transitions of the actor automata sequences into one transition with a non-deterministic choice.

8.2.5 A Simple Example of a MoC CIA

Let's look at a simple example of a MoC CIA. A simple actor model is shown in Figure 8-7. The model consists of two actors, named A1 and A2, each with one input port and one output port. Even though not visible in the figure, the input and output ports of both actors are named In and Out respectively.
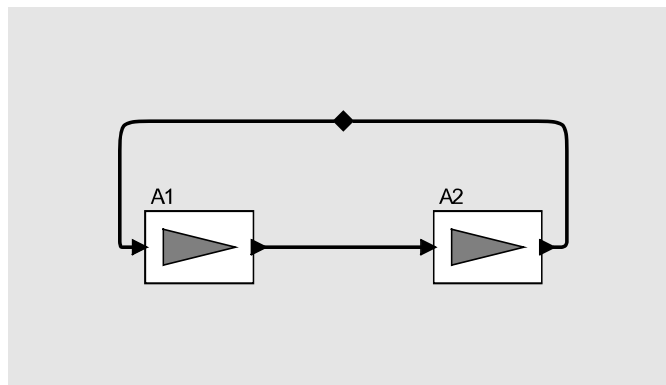
Fig. 8-7: A simple actor model.

Figure 8-8 shows a MoC CIA for the actor model in Figure 8-7. The presented automaton is a typical SDF MoC automaton, since it has a static firing schedule and does not require its actors to check for tokens or for room.
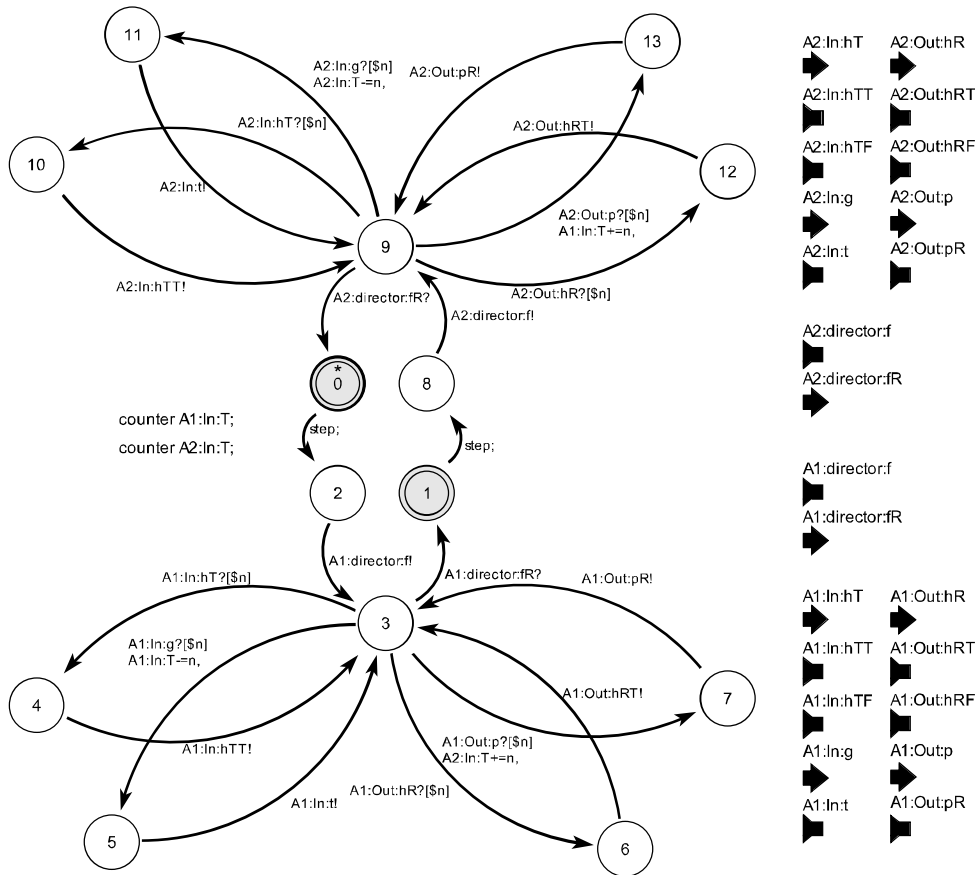
Fig. 8-8: A SDF MoC automaton for the actor model in Figure 8-7.

8.3 Comments

Action Schedules

In Section 8.1.2 , we state that any valid action schedule can be used to schedule a Calflow actor for dataflow MoCs. This comes from the asynchronous message passing that is used in dataflow MoCs, which makes the order of communication acts unimportant. In particular, it does not matter whether a dataflow actor first reads a token from one port and then from another or the other way around.

In MoCs with synchronous message passing, as for example CSP, the order of communication acts however does matter and can in fact usually not be determined statically. The complete CIA of an actor for such a MoC would consist of the set of all legally scheduled CIAs.

Failure States

To explain the importance of failure states, it is best to think of the CIA as a representation of a CAL actor execution. The information specified in the CAL code of an actor corresponds thereby to the CIA we extract of it. In particular, like the CIA we extract, the CAL code does not specify what an actor should do when it reaches a failure state during its execution.

CAL does not specify this, because in general, the correct recovering from a failure state depends on the MoC. For example think of an actor, which needs to consume a token in order to be able to evaluate a guard. If the guard evaluates to false, the actor consumed a token without being able to fire. In this case there are several strategies, on what to do with the consumed token. In a dataflow MoC, the actor may buffer the token internally and use it again when it is fired the next time, in a continuous time MoC on the other hand, the actor should throw away the consumed token, since its value won't be valid anymore at its next firing. Thus, we can think of failure states as place holders for MoC dependant code[2].

Note, the implementation of failure states would also specify the order in which action-matching is determined.

From now on, we ignore failure states and assume the existence of an imaginary magic machine, which does the correct recovering in any case.

---

[2] Lets take this idea one step further and lets assume that, for example, Ptolemy II would provide an interpreter to interpret actors written in CAL. Prior to execute a CAL actor, the interpreter could determine the MoC of the system in which the actor is placed, and could then refine its execution code of the actor, to get the correct, MoC dependant behavior in case a failure state is reached. In such a system, the CAL actor itself would be completely domain polymorphic.

Composing Actor and MoC Automata

The composition and compatibility of two arbitrary Counting Interface Automata is, as we already noted in Chapter 7, not decidable in general. The bad news is, that the composition of actor and MoC automata is also not decidable in general. We will however show in this chapter, how we can exploit certain properties of the structures of generated actor and MoC automata, to make the most important parts of the composition decidable. In fact, the composition and compatibility of the temporal and quantitative aspects of all interfaces in a model will be decidable. The part which remains not decidable, contains the internal token exchange information of the model. In Chapter 10, we will show how to use Petri Nets to analyse the information contained in this part.

9.1 General Structure of the Automata

9.1.1 Structure of Actor Automata

Figure 9-1 shows some examples of typical structures of generated actor automata. An important structure property, shared by all generated actor automata, is that by construction, all paths from one reset state to another are finite, i. e. there is never a loop in the path between two reset states of any actor automaton. Furthermore it has to be noted, that each of these paths between two reset states corresponds to one possible firing of the actor.

9.1.2 Structure of MoC Automata

Figure 9-2 shows some examples of typical structures of generated MoC automata. Generated MoC automata usually have loops and thus infinite paths between reset states. A very important property of these loops on the paths between reset states is however that all of them start with an input transition and can thus be controlled
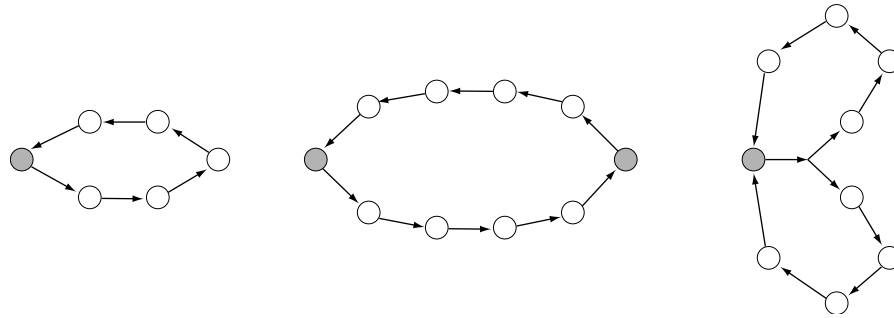
Fig. 9-1:   Typical structures of the generated actor automata. The dark states represent reset states.

from outside. Additionally, as with actor automata and by construction, any path between two reset states of a MoC automaton corresponds to one possible firing of an actor.
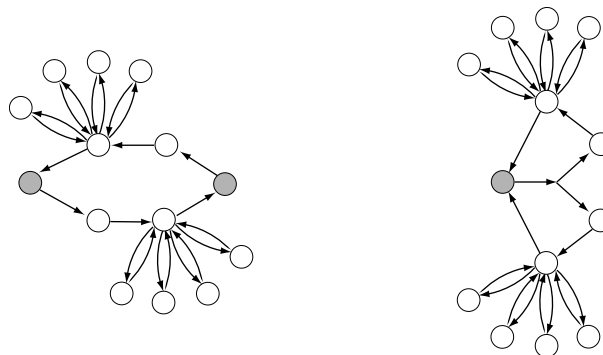
Fig. 9-2:   Typical structures of the generated MoC automata.

9.1.3 Structure of Product Automata

When we build the product of all actor automata and the MoC automaton of a model, we end up with a product automaton which consists of the product of the reset states of all automata, interconnected by finite paths, which correspond to actor firings. In particular, it has to be noted that there are no loops in any path between two reset states of the product.

The reason for this structure comes from the above mentioned properties of the structures of actor and MoC automata. The finiteness of paths between reset states in actor automata and the fact that the loops in any corresponding path in the MoC automaton start with input transitions, results in product paths which are again always finite.

We call the product of all actor automata and the MoC automaton of a model the model product automaton (MPA). Figure 9-3 shows some examples of typical structures of MPAs.
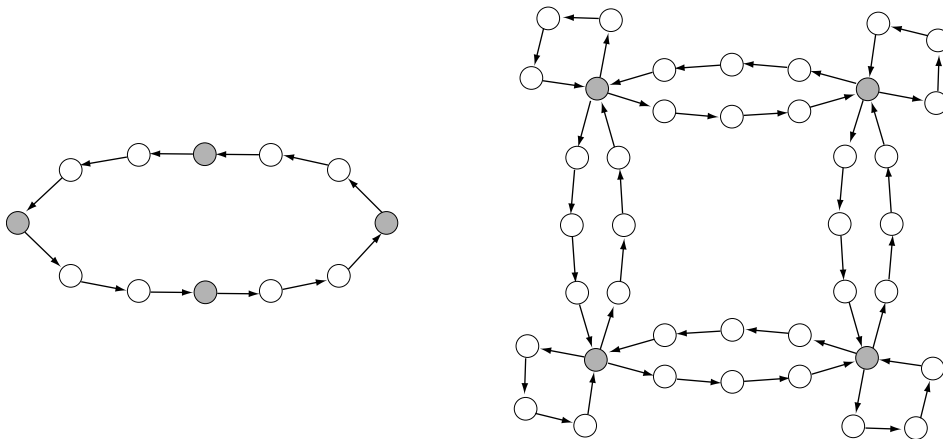
Fig. 9-3:   Typical structures of model product automata.

9.2 Composition Strategy

Unless all paths through a model product automaton have exactly the same global counter assignments, which is the case for statically scheduled SDF models, the composition and compatibility of the model product automaton is not decidable in general.

However, by ignoring the values of all global counters, the composition and compatibility of all paths between two reset states in the model product automaton is decidable, and thus the compatibility of the temporal and quantitative aspects of the interfaces of all actors and the MoC can be ensured.

By doing this, we will end up with a Counting Interface Automaton of the composite model, in which the compatibility of the interfaces of all actors and the MoC is

ensured and in which the information of the internal token exchange rates is contained in the global counter assignments along the paths between reset states.

9.3 Composition Algorithm

The special structure of model product automata, which was discussed above, leads to the fact, that for every state in a path between two reset states, there is only a single path between the path starting reset state and the state itself. Furthermore, since we ignore the values of all global counters, i.e. we leave the global counter variables uninitialized, the context of every reset state contains only uninitialized counter variables. This leads to the property that every state in the model product automaton has only one fix context.

Exploiting this property allows us to use a relatively simple composition algorithm, in which the product and the error states are created at the same time.

We use an algorithm, which recursively expands a product frontier until all reachable product states are analyzed. At every step, the algorithm picks one product state from the frontier, analyses the transitions that can be taken from it, and expands it. Finally, the product states that are reachable from the picked product state get added to the frontier if they were not analyzed yet before.

When the transitions are analyzed, the algorithm ignores any guards, declarations and assignments that contain global counter variables. The information of all declarations and assignments is however still contained in the model product automaton and the correct values of ignored guards is ensured by invariants that are added to the states.

Algorithm
The following pseudo code describes the composition algorithm. In the code, any function $x(t)$ accesses the element x of the transition t [1]. Additionally, we write $t \in u$, if $s(t) = u$, i. e. when the transition starts in u.

---

[1] see Chapter 7 for a description of all elements of a transition.

start

   $frontier = (s_M^{init} \times s_N^{init})$;

   $product = (s_M^{init} \times s_N^{init})$;

   while (frontier ≠ ∅)

      pick a state $(u, v) \in frontier$;

      foreach $(t_u \in u)$

         case $(t_u \in T_M^H)$: *copy(t_u)*;

         case $(t_u \in T_M^I)$:

            case $(a(t_u) \in shared(M, N))$:

               case $(\exists\, t_v \in T_M^O \ with\ t_v \in v\ and\ a(t_v) = a(t_u))$:

                  copyShared(t_u, t_v);

               otherwise: // ignore, this transition can never be activated

            otherwise copy(t_u);

         case $(t_u \in T_M^O)$:

            case $(a(t_u) \in shared(M, N))$

               case $(\exists\, t_v \in T_N^I \ with\ t_v \in v\ and\ a(t_v) = a(t_u))$:

                  copyShared(t_u, t_v);

               otherwise: (t_u, t_v) *is an immediate error state*

            otherwise: copy(t_u);

      endforeach

      foreach $(t_v \in v)$

         case $(t_v \in T_N^H)$: *copy(t_v)*;

         case $(t_v \in T_N^I)$:

            case $(a(t_v) \in shared(M, N))$:

               // ignore, this transition can either never be activated

               // or was already handled above

            otherwise: copy(t_v);

         case $(t_v \in T_N^O)$:

            case $(a(t_v) \in shared(M, N))$

               case $(\exists\, t_u \in T_M^I \ with\ t_u \in u\ and\ a(t_v) = a(t_u))$:

                  // ignore, this transition was already handled above

               otherwise: (t_u, t_v) *is an immediate error state*

            otherwise: copy(t_v);

      endforeach

      case (u, v) *is an error state: frontier = frontier \ (u, v)* ;

otherwise:

$$frontier = (frontier \setminus (u,\ v)) \cup newFrontier;$$

$$product = product \cup newProduct;$$

endwhile

end

copy(t):

case $g(t) \neq false:$

  copy the transition;

  $newFrontier = newFrontier \cup (S'(t) \setminus (S'(t) \cap product));$

  $newProduct = newProduct \cup S'(t);$

otherwise: // ignore

copyShared(t_u, t_v):

copy the shared transition;

case *the transition invariant obtained from this shared transition is false:*

  $(t_u,\ t_v)$ *is a counter error state;*

otherwise:

  $newFrontier = newFrontier \cup$
  $(S'(t_u) \times S'(t_v) \setminus ((S'(t_u) \times S'(t_v)) \cap product))$

  $newProduct = newProduct \cup (S'(t_u) \times S'(t_v))$

Analysis of Actor Models

As we have seen in Chapter 9, a model product automaton (MPA) consists of a number of reset states, interconnected by finite automata sequences, which correspond to actor firings, and the assignments to global counter variables along these sequences contain the internal token exchange information of the composite actor model.

In this chapter, we will show how to extract this internal token exchange information from a MPA, and how to represent it in a Petri Net. This will allow us, to fall back to the well investigated methodologies for Petri Nets, to analyze our composite actor model.

10.1 Extracting Token Exchange Information

10.1.1 Generating Token Exchange Automata (TEA)

To represent the token exchange information of a model product automaton, we first generate an automaton which we call the token exchange automaton (TEA) of a model. The states of this automaton are the reset states of the model product automaton. The transitions correspond to the paths between the reset states of the model product automaton and contain all assignments to global counter variables along these paths. Additionally, they are labelled with the name of the actor they correspond to.

In the TEA, the states represent the states of the composite actor model and each transition represents one possible actor firing, with the assignments on the transition representing the token exchange rates.

Figure 10-1 shows the TEA of the actor model in Figure 10-2. The complete explanation and analysis of this example can be found in Chapter 11.

counter A:In:T;

counter B:In:T;

counter Snk:In1:T;

counter Snk:In2:T;

A;
A:In:T-=1, Snk:In1:T+=1,

A;
A:In:T-=1, Snk:In1:T+=1,

Src;
B:In:T+=1,

B;
B:In:T-=1, Snk:In2:T+=2,

*
0

1

B;
B:In:T-=1, Snk:In2:T+=1,

Src;
A:In:T+=1,

Snk;
Snk:In1:T-=1, Snk:In2:T-=1,

Snk;
Snk:In1:T-=1, Snk:In2:T-=1,
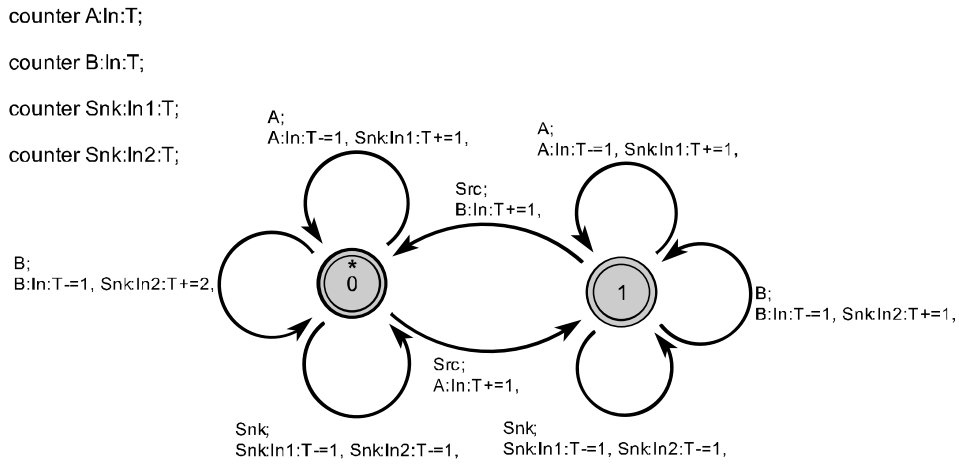
Fig. 10-1: The token exchange automaton of the model in Figure 10-2. The automaton is statefull, since the *Src* actor in the model it is representing is statefull.

Src

A

B

Snk
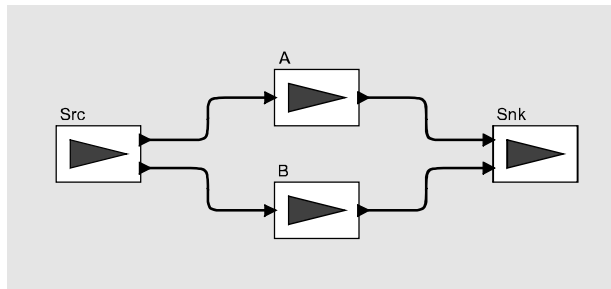
Fig. 10-2: A simple DDF actor model. Actor *Src* is a source actor which is on each firing producing alternatingly one token on its upper and lower output port respectively. Actor *A* consumes one token and produces one token on each firing. Actor *B* consumes one token and produces two tokens on each firing. Actor *Snk* is a sink actor and consumes one token from each input port on each firing.

10.1.2 Generating Token Exchange Petri Nets (TEPN)

Petri Net is a formal and graphical language which is appropriate for modelling systems with concurrency. It was first formally defined 1962 in [16] by Carl A. Petri.

When we generate the Petri Net of a TEA, we logically divide it into a dataflow part and a control part. Figure 10-3 shows the Petri Net of the TEA in Figure 10-2.
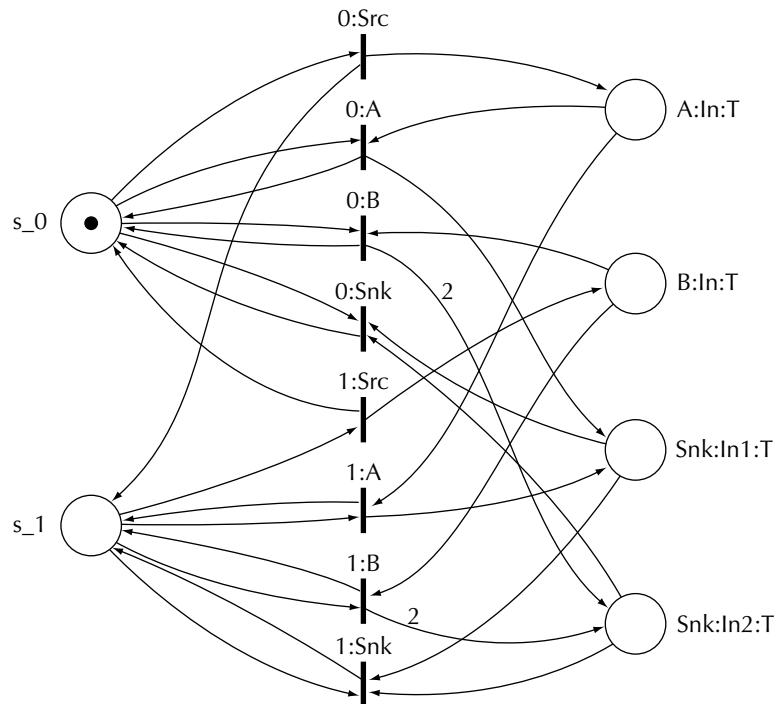
Fig. 10-3:   The token exchange Petri Net of the token exchange automaton in Figure 10-2. The control part of the Petri Net is on the left side and the dataflow part on the right side of the figure. In between are the transitions.

The dataflow part of the generated Petri Net has one place for every global counter variable of the token exchange automaton and the number of markers in each of these places represent the values of the corresponding global counter variables[1].

The control part of the generated Petri Net has one place for every state of the token exchange automaton and one marker in the place which represents the initial state. This marker is used to mark at each time the state in which the model currently is.

There is one transition for every transition of the TEA, each of which represents one possible actor firing in a particular model state of the composite actor model.

The flow relations between the control part of the Petri Net and the transitions determine the fireability of actors in every state of the model. The flow relations between the dataflow part and the transitions on the other hand represent the token

---

[1] Remember that the global counter variables in our models and thus the number of markers in the Petri Net represent the number of tokens in the receiver queue of a channel.

exchange information of every actor firing, where the weights of the flow relations correspond to the number of tokens exchanged during each firing.

10.2 Analyzing Token Exchange Behavior

Petri Nets are a well established research area with a vast amount tools are available. In this context however, we will only concentrate on three properties of Petri Nets, which are of particular interest for the analysis of composite actor models.

- *Security* in the TEPN corresponds to the boundedness of receivers in the actor model.

- *Reversibility* in the TEPN corresponds to the availability of a legal firing sequence[2] for the actor model.

- *Liveliness* in the TEPN corresponds to the liveliness of the actor model.

10.2.1 Security

The security of a Petri Net can be determined using the reachability tree analysis. In the reachability tree, every insecure place will eventually become represented with a infinity symbol ∞. A Petri Net is secure, if its reachability tree does not contain such an infinity symbol.

If the TEPN is secure, then the buffer lengths of all receivers in the actor model are bounded.

10.2.2 Reversibility

For secure Petri Nets, reversibility can be determined again using the reachability tree analysis. If the Petri Net is not secure, the existence of a transition invariant is a required but not sufficient condition for the reversibility of the Petri Net.

The reversibility of the TEPN is a requirement for the existence of a legal firing sequence. It is however not sufficient, since the TEPN ignores all action guards and can thus decide to execute actions that would maybe not be executable under certain conditions at runtime.

---

[2] A legal firing sequence is a firing sequence, which can be executed infinitely, without leading to deadlock or buffer overflow.

From the non-existence of a transition invariant however, we can conclude that there exists for sure no legal firing sequence for the actor model.

10.2.3 Liveliness

For secure Petri Nets, liveliness can be determined using the reachability tree analysis.

The deadlock freeness of the TEPN is a requirement for the deadlock freeness of the actor model. As with for the reversibility above, it is however not sufficient, again because of the guards.

Part III
Case Studies

Case Studies

11.1 Component Interface Compatibility

11.1.1 An SDF Actor in a DDF Model of Computation

The automaton at the top of Figure 11-1 shows the CIA of a simple SDF actor, which produces one token on its output port and consumes one token from its input port. Since the shown actor is implemented for a SDF MoC, it does neither check for tokens nor for room.

Just below is the CIA of a DDF MoC for the same actor. Since the DDF MoC requires all its actors to check for tokens prior to consuming them, the interface of the SDF actor should be incompatible with the one of the DDF MoC.

The product of the SDF actor with the DDF MoC, annotated with the context of each state, is shown at the bottom of Figure 11-1. When the SDF actor tries to get a token without prior checking, the guard on the corresponding input transition of the MoC automaton will be false[1] and the transition will therefore not be active. This leads to the counter error state in 2:4 of the product.

After pruning out all states along the unconditional execution fragment which leads to the error state, as well as after pruning out all unreachable states, the composition of the SDF actor and the MoC automaton will be empty, showing as expected their interface incompatibility.

11.1.2 An Illegal Actor in a DDF Model of Computation

At the top Figure 11-2 is the CIA of a DDF actor, with only one input port. From the automaton it can be seen, that the implementation of the actor is flawed, since it checks for the availability of 4 tokens, but it consumes all in all 5 tokens.

---

[1] Note, that Ramp:In:cT is a local variable and is therefore not ignored during composition. Since it is uninitialized, its comparison with 1 is always false.
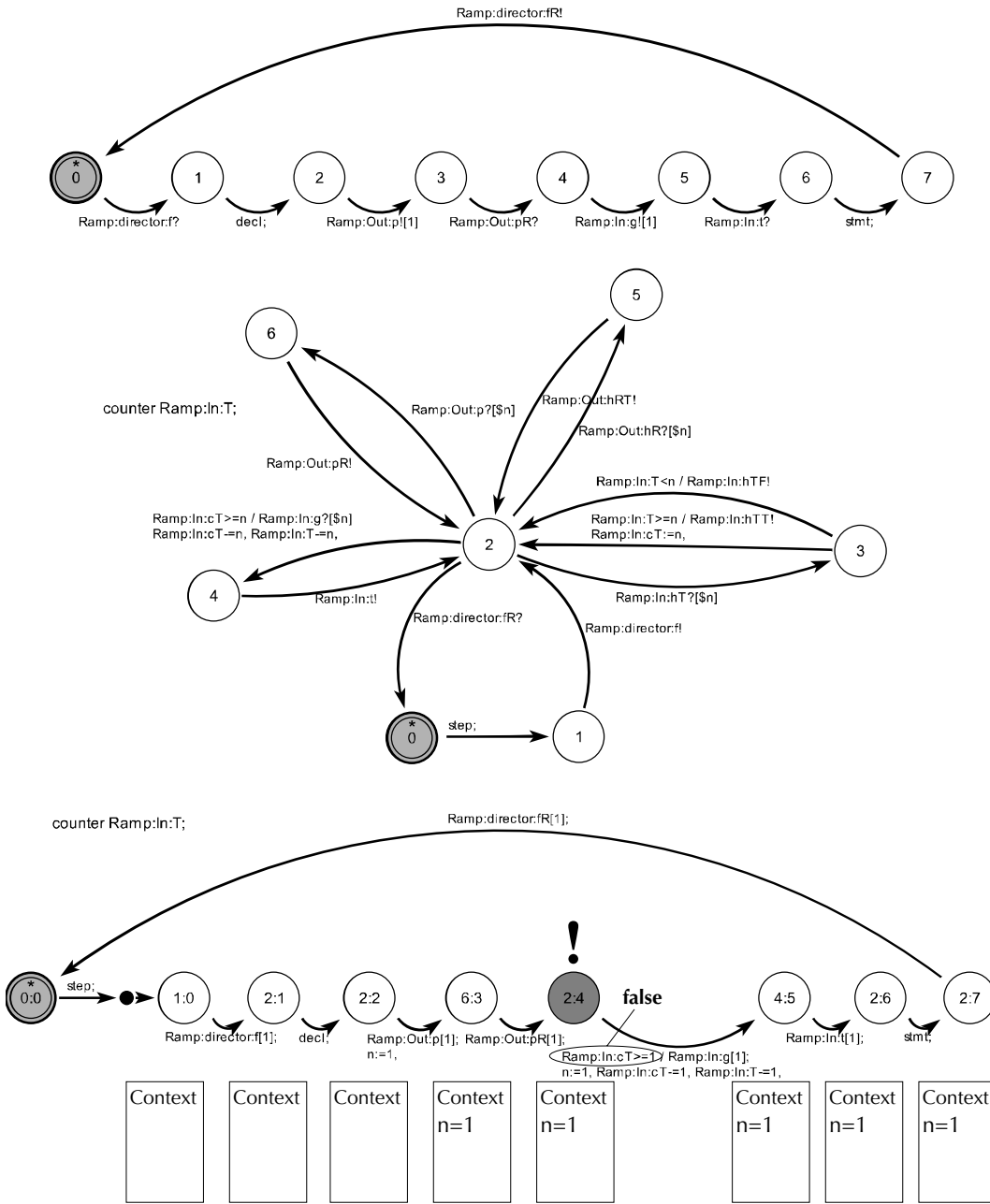
Fig. 11-1: A SDF actor automaton, a DDF MoC and their product. State 2:4 is a counter error state in the automaton product. After pruning, the composition will be empty, which shows the interface incompatibility. Remember, internal transitions have no quantities, the string [n] is part of their name.

In the product of the DDF actor with the DDF MoC, which is shown at the bottom of Figure 11-2, it can be seen, that the consumption of the first two tokens by the actor are compatible with the MoC automaton. But the consumption of another three

tokens, of which the actor checked only two, leads to a counter error state in state 2:5 of the product.

After pruning, the composition of the actor automaton with the MoC automaton will be empty, showing again interface incompatibility.
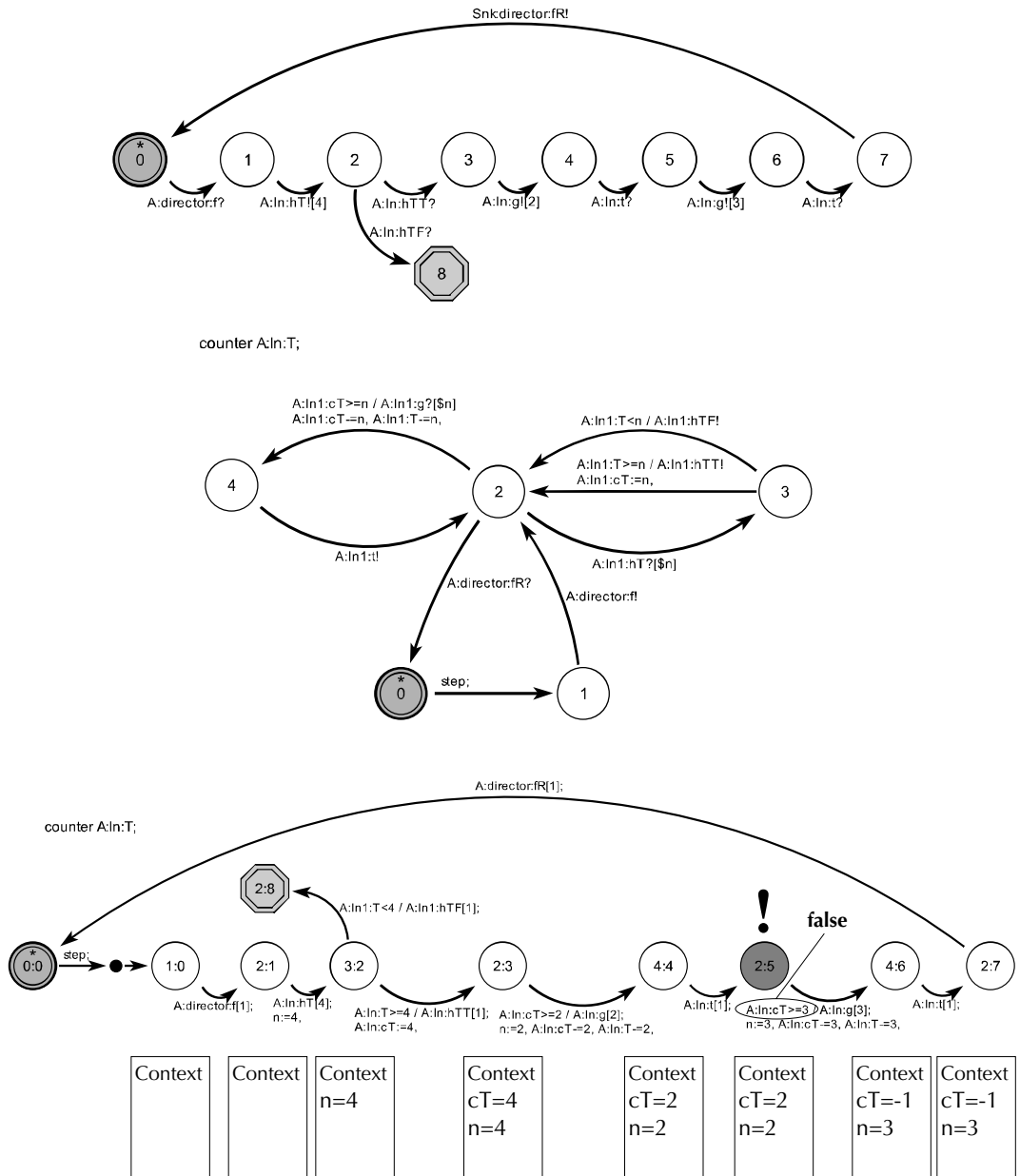
Fig. 11-2:   A DDF actor automaton, a DDF MoC and their product. State 2:5 is a counter error state in the automaton product. After pruning, the composition will be empty, which shows the interface incompatibility.

11.2 Analysis of Dataflow Actor Models

11.2.1 An Actor Model with Incompatible Dataflow Rates

In this case study, we will go through the complete process of CIA generation, composition and subsequent analysis of the actor model we already introduced in Chapter 10 and which is shown again in Figure 11-3.
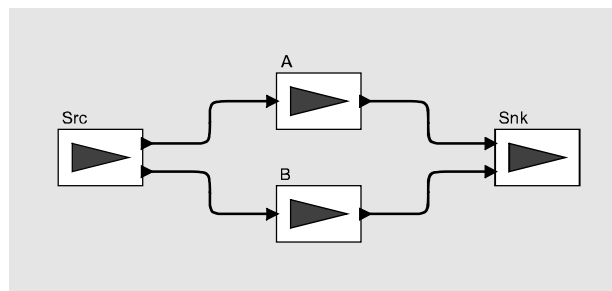
Fig. 11-3:   The actor model.

The Actor Model

This simple actor model is composed of four different actors, each of which is explained in one of the following sections.

Actor Src

This actor has no input ports and two output ports. It is a source in our model. Each time it is fired, it produces alternatingly one token on either its upper or lower output port. The CAL code of this actor is shown below. An action selector ensures the alternating choice of the two actions.

```
1: actor Src () ==> T Out1, T Out2;
2:    A1:action ==> [a], [] end
3:    A2:action ==> [], [a] end
4:    selector
5:        ([A1][A2])*;
6:    end
7: end
```

Actor A

This actor has one input port and one output port. Each time it is fired, it consumes one token from its input port and produces one token on its output port. In fact, this actor is an identity actor.

```
1: actor A () T In ==> T Out;
2:    A1:action [a] ==> [a] end
3: end
```

Actor B

This actor has also one input port and one output port. Each time it is fired, it consumes one token from its input port and produces two tokens on its output port.

```
1: actor B () T In ==> T Out;
2:    A1:action [a] ==> [a, a] end
3: end
```

Actor Snk

This actor has two input ports and no output port. It is a sink in our model. Each time it is fired, it consumes one token from each of its two input ports.

```
1: actor Snk () T In1, T In2 ==> ;
2:    A1:action [a], [b] ==> end
3: end
```

Step 1: Generating Actor Automata

Since none of the actors in the model have any data dependencies except a simple input-output dependency, the Calflow representation of all of these actors is very simple and is omitted here. Figure 11-4 shows the actor automata of all four actors.

Note, that the actor automaton of the *Snk* actor has two states because of its actor selector. Furthermore note, how the information of the production of two tokens is contained in the put transition of actor *B*.

Step 2: Generating MoC Automata

Figure 11-5 shows a MoC automaton for the actor model. The automaton has four global counter variables, each of which is representing the token queue length in one of the four receivers in the model.

The MoC in this automaton does not give any firing guarantees an thus requires all actors to check for tokens, prior to consumption. But it does not require the actors to check for room. Furthermore it has no static firing schedule.
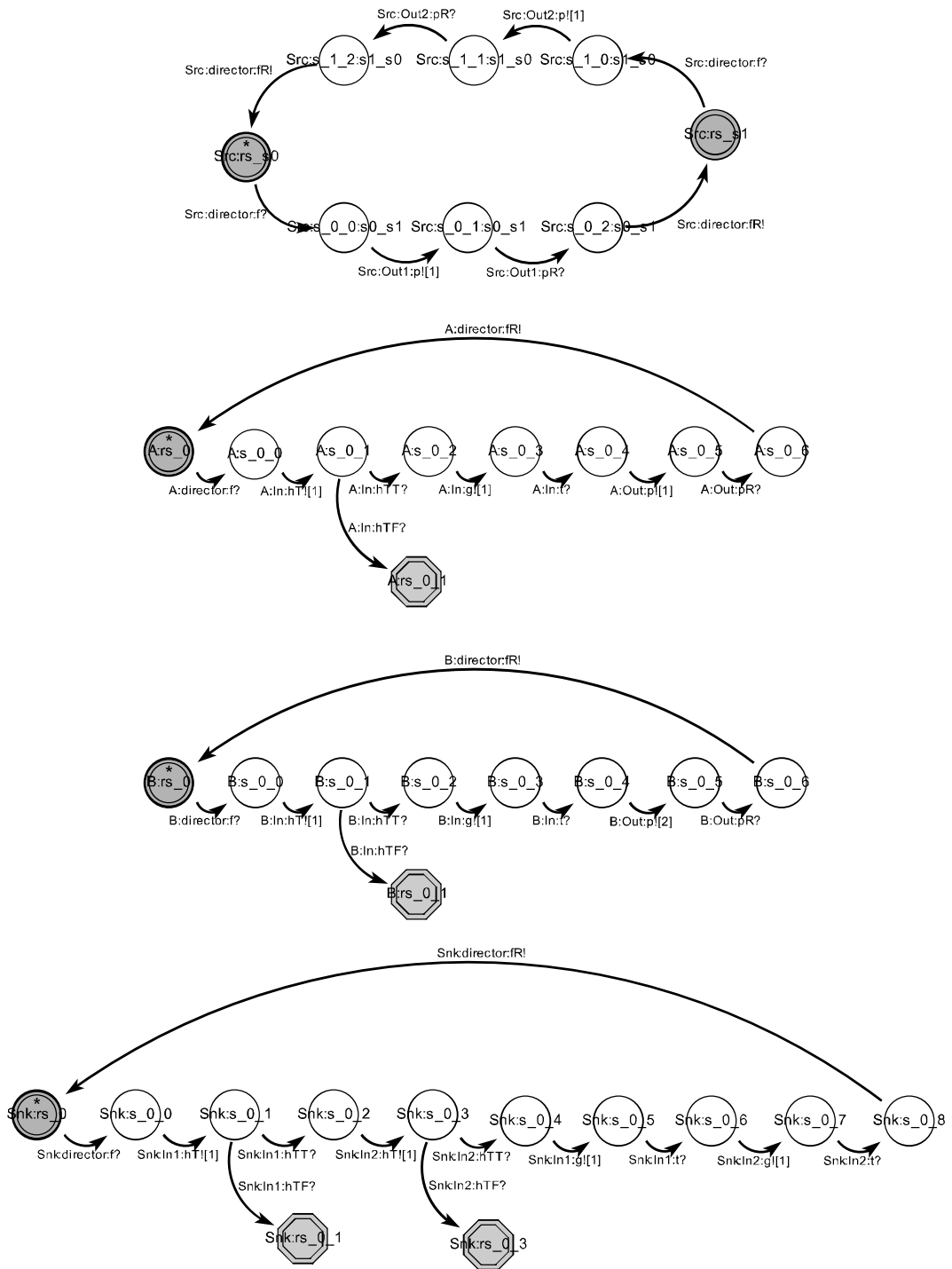
Fig. 11-4:   The CIA of all actors in the model.
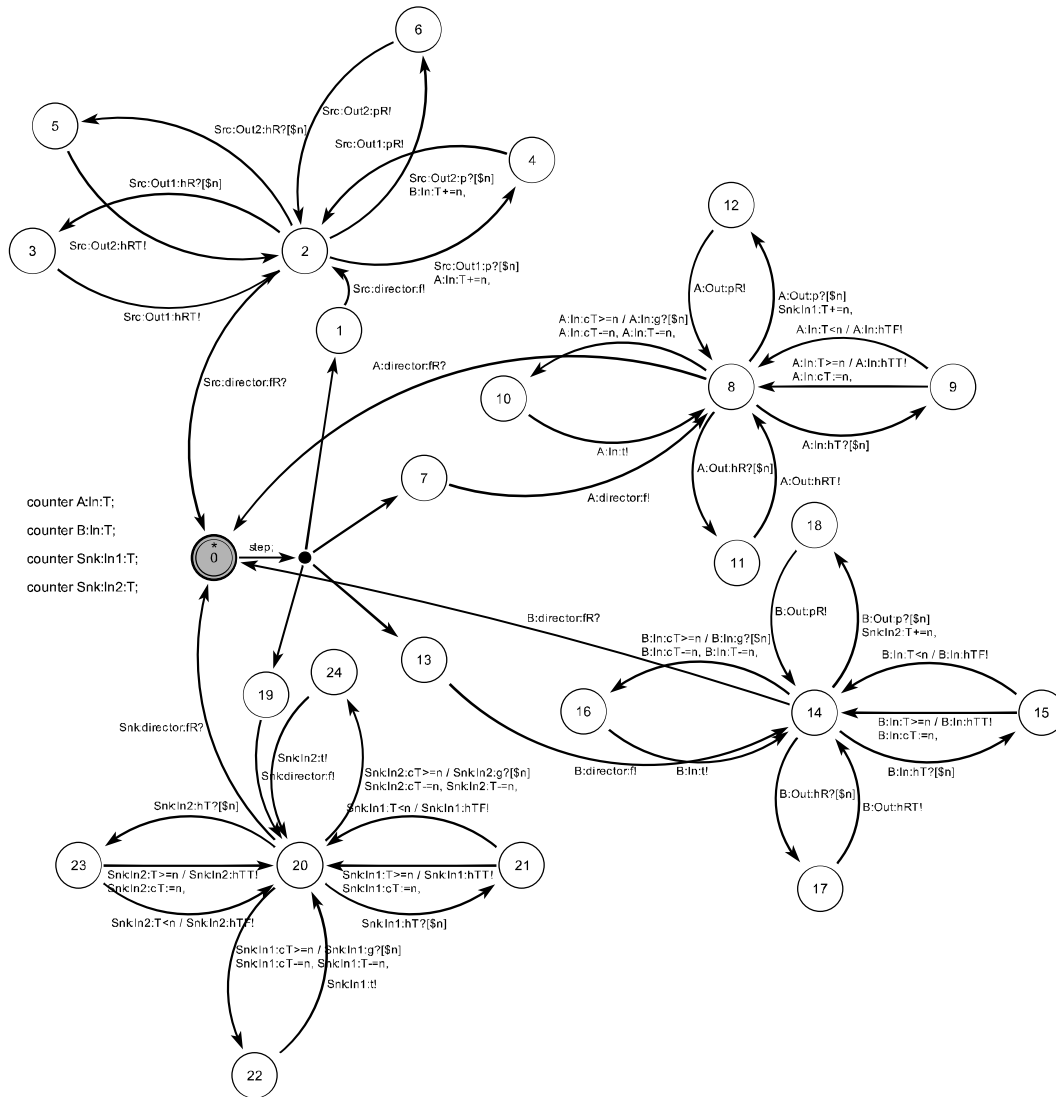
Fig. 11-5:   The CIA of the MoC. State 2 is the actor execution state of the *Src* actor, state 8 the one of actor *A*, state 14 the one of actor *B*, and state 20 the one of the *Snk* actor.

Step 3: Generating Model Product Automata (MPA)

Figure 11-6 shows the model product automaton, which is the result of composing iteratively every actor automaton with the MoC automaton of the model.

The MPA has two reset states, representing the state in which the *Src* actor of the model is. In both reset states, the MoC may choose to fire any actor, and when firing the *Src* actor, a state change occurs.

Note, when we can successfully generate the MPA, this proves the component interface compatibility of all actors and the MoC in the model.
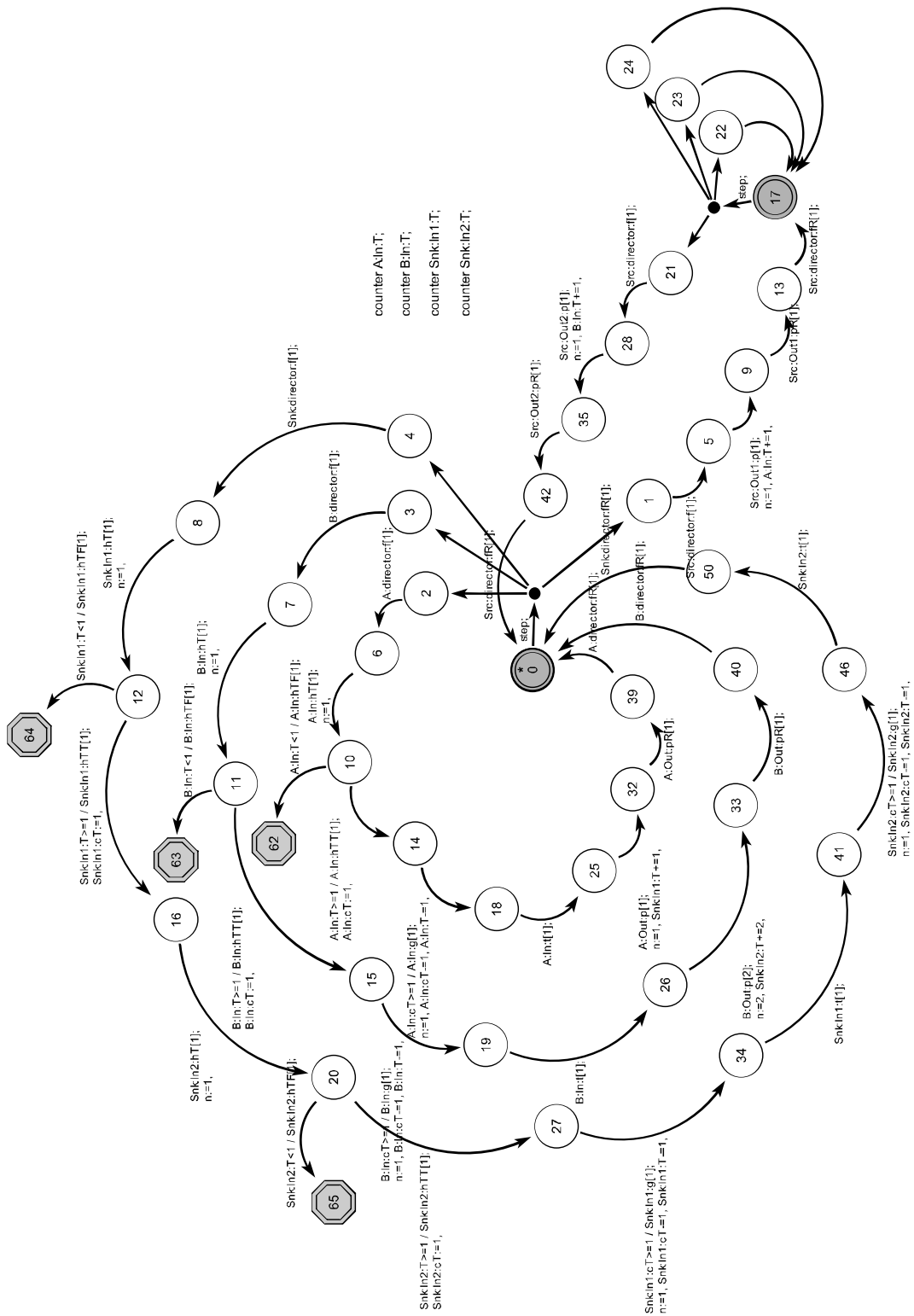
Fig. 11-6: The MPA of the actor model. Note, that the states 22, 23 and 24 would be followed by the same automaton sequences as the states 2, 3 and 4 respectively. For a better visibility, these additional cycles are omitted in the figure.

Step 4: Generating Token Exchange Automata (TEA)

Figure 11-7 shows the token exchange automaton, which is obtained by extracting the token exchange information of the MPA.

In the TEA the two states of the actor model are again visible. The transitions between the two states correspond to the firing of the *Src* actor, while the firing of all other actors does not lead to a state change.

counter A:In:T;

counter B:In:T;

counter Snk:In1:T;

counter Snk:In2:T;

A;
A:In:T-=1, Snk:In1:T+=1,

A;
A:In:T-=1, Snk:In1:T+=1,

Src;
B:In:T+=1,

B;
B:In:T-=1, Snk:In2:T+=2,

B;
B:In:T-=1, Snk:In2:T+=1,

Src;
A:In:T+=1,

Snk;
Snk:In1:T-=1, Snk:In2:T-=1,

Snk;
Snk:In1:T-=1, Snk:In2:T-=1,
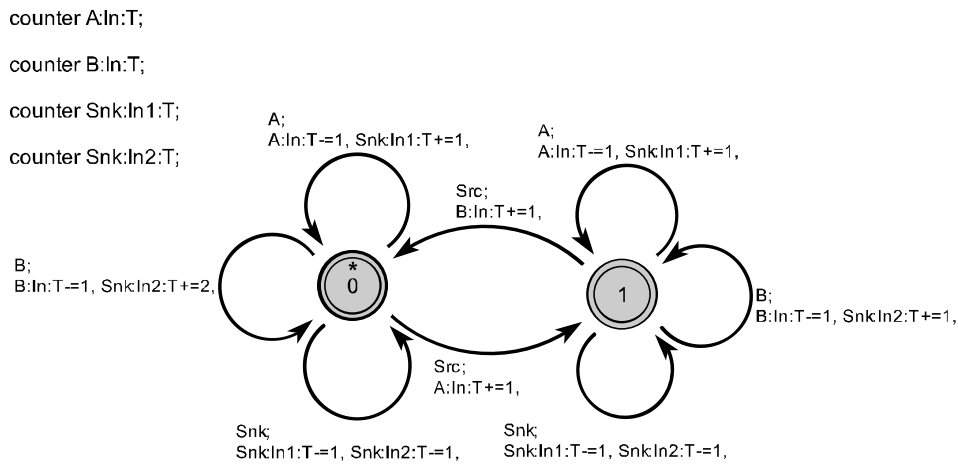
Fig. 11-7:   The TEA of the actor model.

Step 5: Generating Token Exchange Petri Nets (TEPN)

Figure 11-8 shows the token exchange Petri Net which is representing the token exchange information of the TEA.

Step 6: Analysis of the Token Exchange Petri Net

Linear Algebra Analysis

For the linear algebra analysis, we represent the TEPN in its structural matrix. We then calculate the transition invariant of the structural matrix. A is the structural matrix of the TEPN and I_T the transition invariant.

Fig. 11-8:   The TEPN of the actor model.

$$A = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 2 & -1 & 0 & 0 & 2 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix}$$

$$I_T = 0$$

Reachability Tree Analysis

In general, any DDF model with source actors and without schedule will not be secure. The source actor can be fired infinitely often, and thus the receiver buffers of any actor port connected to the source actor will be unbounded. Since this unboundedness will often flood over big parts of the model, the reachability tree will usually not show interesting information.

To overcome this problem, we introduce a virtual trigger signal to the model, such that while building the reachability tree, the transition which corresponds to a source actor, may only be fired when a trigger signal occurs. We then send a trigger

signal each time when no other transitions than the source actor transition are active in the TEPN. Moreover, we will only start analyzing the reachability tree after the initial trigger signal. We will call a reachability tree that was built under these constraints a triggered reachability tree.

This strategy seems to work relatively fine for DDF models in which at most one source action is enabled in every state of the model. At this moment, it is however not clear, how this strategy could be extended to work with more than one sources.

Figure 11-9 shows the triggered reachability tree of the TEPN.

(s_0 | s_1 | A:In:T | B:In:T | Snk:In1:T | Snk:In2:T)

$\boxed{1\ 0\ 0\ 0\ 0\ 0}$

↓ trigger; 0:Src

0 1 1 0 0 0

↓ 1:A

0 1 0 0 1 0

↓ trigger; 1:Src

1 0 0 1 1 0

↓ 0:B

1 0 0 0 1 2

↓ 0:Snk

1 0 0 0 0 1

↓ trigger; 0:Src

0 1 1 0 0 ∞

↓ 1:A

0 1 0 0 1 ∞

↓ trigger; 1:Src

1 0 0 1 1 ∞

trigger; 0:Src   ↓ 0:B

1 0 0 0 1 ∞

↓ 0:Snk
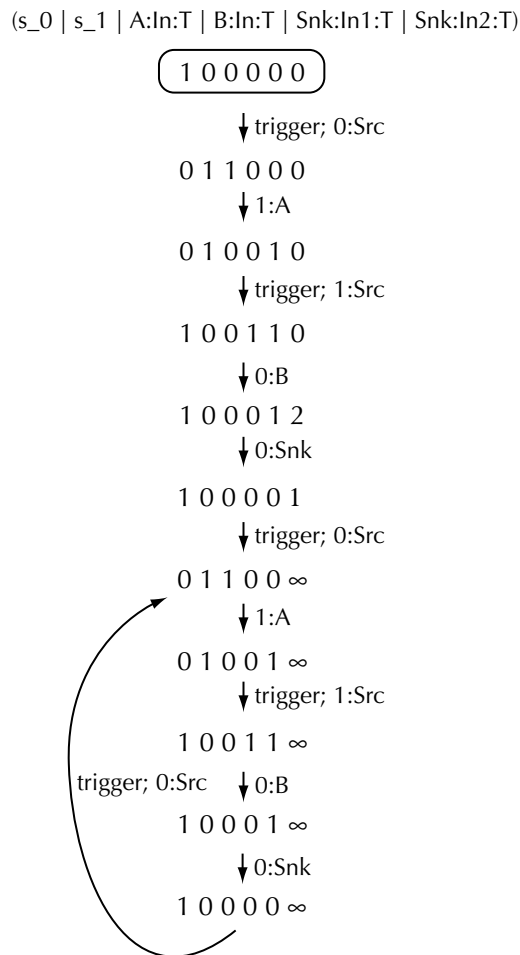
1 0 0 0 0 ∞

Fig. 11-9:   The triggered reachability tree of the TEPN. Note that we start analyzing the tree only *after* the initial trigger signal, thus the state (100000) is not included in the analysis of security.

Discussion

From the non-existence of a non-zero transition invariant, we know that the TEPN is not reversible and that therefore no legal firing sequence exists for the actor model.

From the triggered reachability tree, we can show that the receiver of port *In2* of the *Snk* actor is unbounded, and that the actor model has firing sequences without deadlock.

An Improved Actor Model

From the obtained results, and from looking at the actor model, it is clear that actor *B* introduces a problem to the actor model, by duplicating the token rate of its input port.

To fix this problem, we replace actor *B* with a copy of actor *A*, such that both paths between the source and the sink have the same token flow rate. Note, we could also replace actor *A* with a copy of actor *B*.

A' is the structural matrix of the new TEPN and $I_T^{T'}$ the transpose of a valid transition invariant.
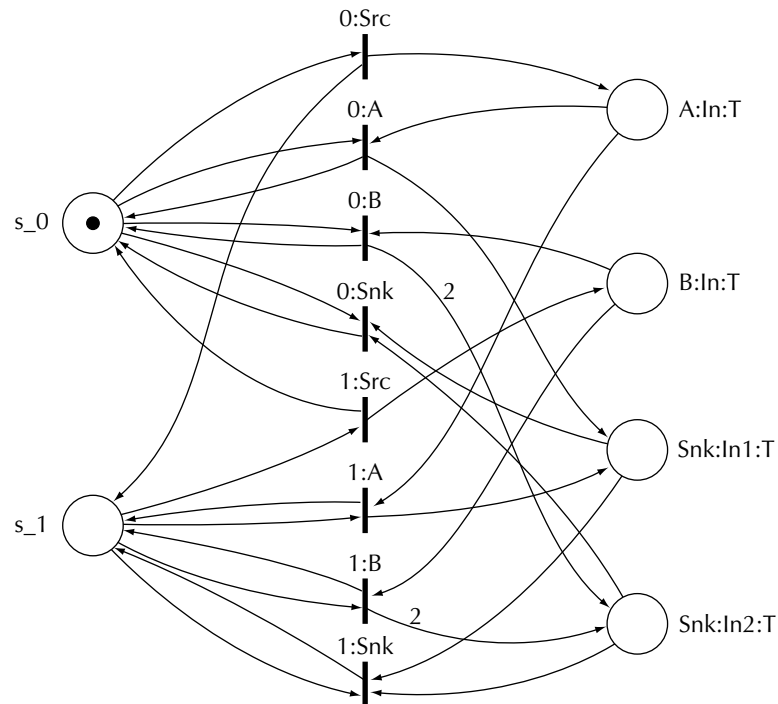
$$A' = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix}$$

$$I_T^{T'} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Figure 11-10 shows the triggered reachability tree of the improved actor model.

The existence of a non-zero transition invariant already fulfills a requirement for the existence of a legal firing sequence. And from the triggered reachability tree, we can see, that $Src \rightarrow A \rightarrow Src \rightarrow B \rightarrow Snk$ is a legal firing sequence that leads to no deadlock and to boundedness of the system.

11.2.2 An Actor Model with Feedback

In this case study, we will analyse an actor model with feedback loops. We will jump over the process of CIA generation and composition, and will start directly from the TEA of the actor model. The CIA generation and composition however work exactly the same as in the previous case study.

(s_0 | s_1 | A:In:T | B:In:T | Snk:In1:T | Snk:In2:T)
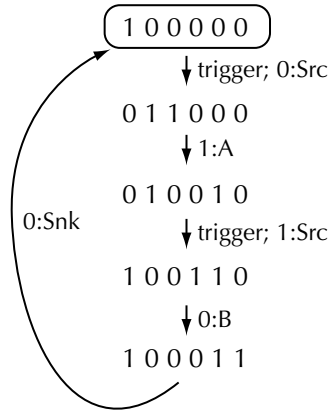
Fig. 11-10: The triggered reachability tree of the improved actor model.

The Actor Model

The actor model of this case study is shown in Figure 11-11 and is again composed of four different actors, each of which is explained in the following sections.
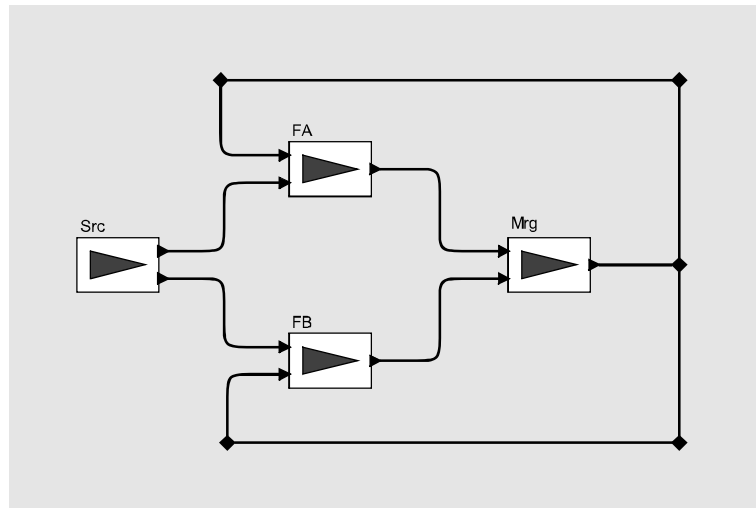
Fig. 11-11: The actor model with feedback.

Actor Src

This is the same actor as in the previous case study.

Actor FA

This actor has two input ports and one output port. Each time it is fired, it consumes one token from each of its input ports and produces one token on its output port.

```
1: actor A () T In1, T In2 ==> T Out;
2:    A1:action [a, b] ==> [a] end
3: end
```

Actor FB

This actor is a copy of the same actor as *FA*.

Actor Mrg

This actor has two input ports and one output port. It implements a non-deterministic merge function. Each time it is fired, it consumes one token from one of its two input ports and produces one token on its output port.

```
1: actor Mrg () T In1, T In2 ==> ;
2:    A1:action [a], [] ==> [a] end
2:    A2:action [], [a] ==> [a] end
3: end
```

The Token Exchange Automata

Figure 11-12 shows the TEA of the actor model, which was obtained by extracting the token exchange information of the model product automaton from the actor model.

Because of the statefull *Src* actor, the TEA has again two states.

Analysis of the Token Exchange Petri Net

Linear Algebra Analysis

A is the structural matrix and I_T the transition invariant of the TEPN which represents the token exchange information of the above TEA.

$$
A = \begin{bmatrix}
1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 1 & 0 & -1 & 0 & 1 & 1 \\
0 & 0 & -1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\
0 & 0 & -1 & 1 & 1 & 0 & 0 & -1 & 1 & 1 \\
0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 & -1 & 0 \\
0 & 0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 & -1 \\
-1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

MrgA;
Mrg:In1:T-=1, FA:In2:T+=1, FB:In2:T+=1

MrgA;
Mrg:In1:T-=1, FA:In2:T+=1, FB:In2:T+=1

FA;
FA:In1:T-=1, FA:In2:T-=1, Mrg:In1:T+=1

FA;
FA:In1:T-=1, FA:In2:T-=1, Mrg:In1:T+=1

counter FA:In1:T;

counter FA:In2:T;

counter FB:In1:T;

counter FB:In2:T;

counter Mrg:In1:T;

counter Mrg:In2:T;

Src;
FB:In1:T+=1,

Src;
FA:In1:T+=1,

FB;
FB:In1:T-=1, FB:In2:T-=1, Mrg:In2:T+=1

FB;
FB:In1:T-=1, FB:In2:T-=1, Mrg:In2:T+=1

MrgB;
Mrg:In2:T-=1, FA:In2:T+=1, FB:In2:T+=1

MrgB;
Mrg:In2:T-=1, FA:In2:T+=1, FB:In2:T+=1

Fig. 11-12: The TEA of the actor model.

$$I_T = 0$$

Reachability Tree Analysis

Figure 11-13 shows the triggered reachability tree of the TEPN. Note, that an initial token must be present on the feedback loops.

Discussion

From the non-existence of a non-zero transition invariant, we know again, that no legal firing sequence exists for the actor model.

From the triggered reachability tree, we can show that the receivers of port *In2* of both actors *FA* and *FB* are unbounded, and that the actor model has firing sequences without deadlock.

(s_0 | s_1 | FA:In1:T | FA:In2:T | FB:In1:T | FB:In2:T | Mrg:In1:T | Mrg:In2:T)

$$\boxed{1\ 0\ 0\ 1\ 0\ 1\ 0\ 0}$$

↓ trigger; 0:Src

0 1 1 1 0 1 0 0

↓ 1:FA

0 1 0 0 0 1 1 0

↓ 1:MrgA

0 1 0 1 0 2 0 0

↓ trigger; 1:Src

1 0 0 1 1 2 0 0

↓ 0:FB

1 0 0 1 0 1 0 1

↓ 0:MrgB

1 0 0 2 0 2 0 0

↓ trigger; 0:Src

0 1 1 ∞ 0 ∞ 0 0

↓ 1:FA

0 1 0 ∞ 0 ∞ 1 0

↓ 1:MrgA

0 1 0 ∞ 0 ∞ 0 0

↓ trigger; 1:Src

1 0 0 ∞ 1 ∞ 0 0

trigger; 0:Src   ↓ 0:FB

1 0 0 ∞ 0 ∞ 0 1

↓ 0:MrgB

1 0 0 ∞ 0 ∞ 0 0
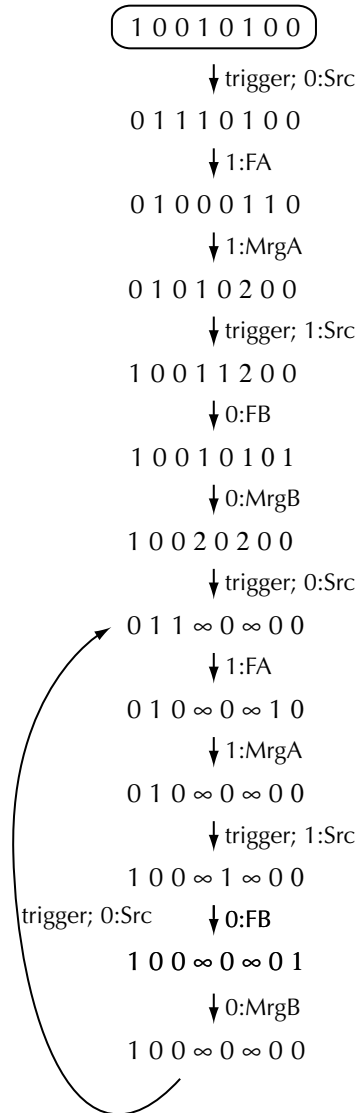
Fig. 11-13: The triggered reachability tree of the TEPN.

An Improved Actor Model

The unboundedness in the above model occurs from the merge actor, which consumes only one token that was produced by either *FA* or *FB*, but produces one token for both *FA* and *FB*.

To fix this problem, we replace the *Mrg* actor with another merge actor which consumes a token from both input ports and produces one token on its output port. In fact, this new actor is a copy of *FA* and *FB*.

A' is the structural matrix of the new TEPN and $I_T^{T'}$ the transpose of a valid transition invariant.

$$A' = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix}$$

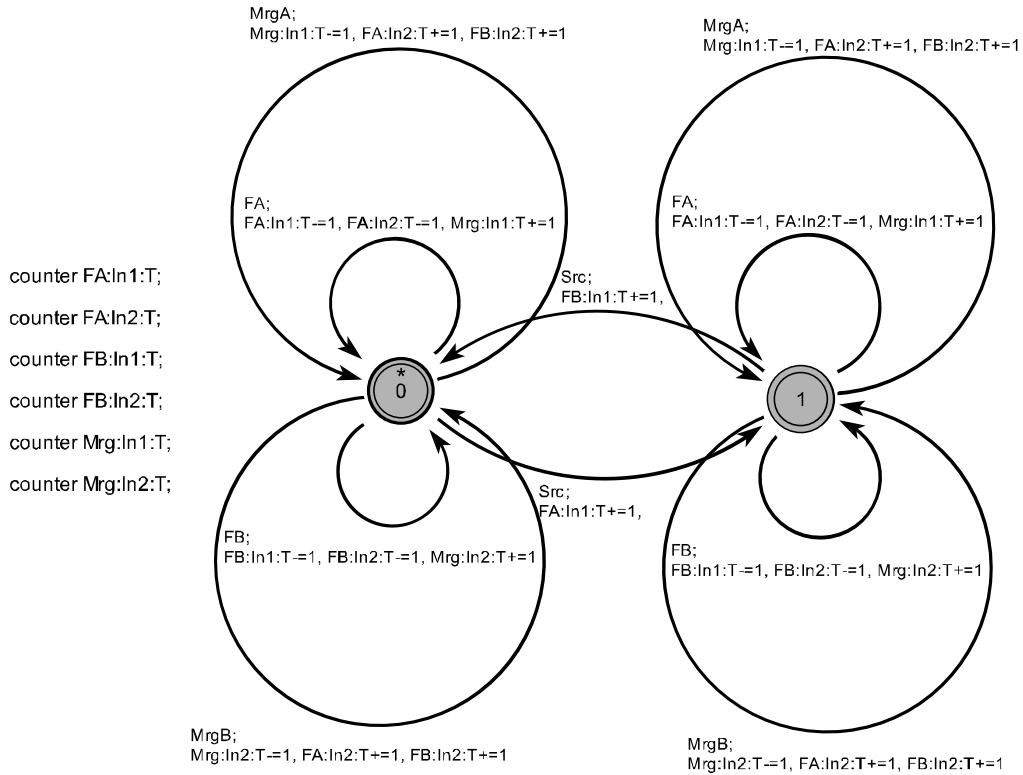$$I_T^{T'} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Figure 11-14 shows the triggered reachability tree of the improved actor model.

(s_0 | s_1 | FA:In1:T | FA:In2:T | FB:In1:T | FB:In2:T | Mrg:In1:T | Mrg:In2:T)
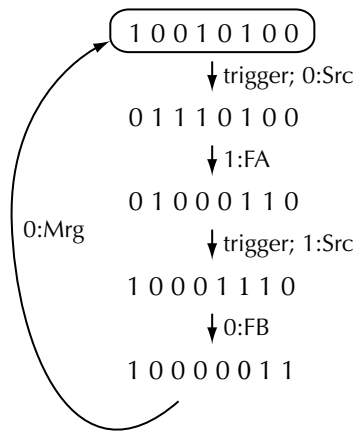
Fig. 11-14: The triggered reachability tree of the improved actor model.

From the triggered reachability tree, we can see, that $Src \rightarrow FA \rightarrow Src \rightarrow FB \rightarrow Mrg$ is a legal firing sequence that leads to no deadlock and to boundedness of the improved actor model.

The presented improved actor model fixes the problems of the original actor model. However, theoretically it would have been sufficient to add the action of the new merge actor as a third action to the original *Mrg* actor. With such a model we

however reach the limits of the analysis with triggered reachability trees. To produce enough tokens to fire the third action of the extended *Mrg* actor, we would need to fire the *Src* actor in a state when the *Mrg* actor would still be fireable using one of the original actions. This improved model however seems also to be problematic with other strategies for DDF directors.

Part IV
Conclusions

Chapter 12

Conclusions

12.1 Conclusions

We proposed a new interface theory, Counting Interface Automata (CIA), which is capable of counting with natural numbers. We used CIA to represent the interface information of actors, and its counting capabilities allowed us to capture both, the temporal aspects of the actor interface, as well as its token exchange rates. The captured information proved to be expressive enough for interesting analysis of actor systems based on the CIA of their actors and the MoC.

We then presented a solution for automatic extraction of CIAs from actors written in CAL. For this task, we developed an intermediate format, called Calflow, from which we actually extracted the CIAs.

Additionally to the automatic extraction of actor CIAs, we presented a method to create a CIA, which represents the interface of a system environment for SDF and DDF MoCs. The counting capabilities of CIA allowed us, to capture the actor connection information of the system, such that the system environment CIA could be used as glue between the CIAs of all its actors.

Since the actor CIAs and the system environment CIA could capture the token exchange rates and the actor connection information respectively, they could together capture the complete token exchange information of the system. We showed, that by composing all CIAs we could get a single CIA, which contained exactly this token exchange information of the composite system.

The token exchange information captured in this CIA could easily be transformed into a Petri Net, which allowed us to fall back to the well investigated methods for Petri Nets, to analyze the composite actor model.

12.2 Outlook

The material presented in this thesis raises a number of topics that would be interesting for further research.

For the future work on the presented material it would be interesting to define automata generators for other MoC's than DF. Some basic steps were already made towards representing a DE MoC as a CIA. In this case, counter values were used to represent both data tokens as well as discrete time units.

The presented automata generator for DDF creates a very basic controller, in which in fact the complete actor firing strategy is left open. However, the ability of automatically obtaining the token rates from all actors in a model, could lead towards sophisticated DDF controllers, which would take this token rate information into account when determining their firing strategy.

In the presented work, we always built closed actor systems, i.e. systems without any external ports. In open systems with external ports, the model product automaton (MPA) would capture the interface description of the composite actor. This would allow us to generate the CIA of a composite actor by generating the MPA using the CIAs of its atomic actors. For composite actors with an SDF MoC, this is a straightforward task and can already be done with the presented techniques. However, for other MoCs it seems to be more difficult to obtain a reasonable CIA for the composite actor. It would be interesting to investigate the problems that occur with other MoCs, to obtain techniques that would enable us to generate composite actor CIAs for them.

Finally, Calflow seems to be an ideal intermediate format for code generation of CAL actors. Its explicit representation as dataflow graph allows to easily generate execution schedules, which can be optimized towards different application areas. Furthermore, the concept of failure states could be interesting to formalize MoC dependant implementations of CAL actors.

Part V

Appendix

Calflow

A.1 Transforming CAL into Calflow

We can represent a CAL actor as tuple

$$P_{Cal} = (T,\ D_{Param},\ D_{Port},\ D_{State},\ N,\ S_{Init},\ A,\ \varphi)$$

consisting of the following components.

- T is a set of *type parameters.*

- D_{Param} is a set of *parameter declarations.*

- D_{Port} is a set of *port declarations.*

- D_{State} is a set of *state variable declarations.*

- N is a set of *invariants.*

- S_{Init} is a set of *initialization statements.*

- A is a set of *actions.*

- φ is an *action selector.*

We can also represent a Calflow actor as a tuple

$$P_{Calflow} = (T,\ D'_{Param},\ D'_{Port},\ D'_{State},\ N',\ S'_{Init},\ A',\ \varphi')$$

consisting of the same component types as a CAL actor.

We can then transform a CAL actor into a Calflow actor by separately transforming every component and by unifying them.

$$(P_{Cal} \Rightarrow P_{Calflow}) = \left(\bigcup_{\forall comp} (comp \Rightarrow comp') \right)$$

For all components except the actions, this transformation turns out to be a trivial equality operation.

A.1.1 Transforming Actions

We can represent a Cal action as a tuple $A = (I, O, G, D, S)$ consisting of the following components.

- I is a set of *input patterns*.

- O is a set of *output expressions*.

- G is a set of *guards*.

- D is a set of *variable declarations*.

- S is a set of *statements*.

Further, we can represent a Calflow action as a tuple $A' = (\Pi, \Delta, \tau)$ consisting of the following components.

- Π is a set of *atomic steps*.

- Δ is a set of *dependencies*.

- τ is an *action schedule*.

With $\Pi = \Pi_{IG} \cup \Pi_I \cup \Pi_{OG} \cup \Pi_O \cup \Pi_G \cup \Pi_D \cup \Pi_S$, where Π_{IG}, Π_I, Π_{OG}, Π_O, Π_G, Π_D and Π_S are the mutual disjoint sets of *input guard, input, output guard, output, guard, decl* and *stmt atomic steps* respectively. And $\Delta = \Delta_D \cup \Delta_C$, where Δ_D and Δ_C are the sets of *data dependencies* and *constraint dependencies* respectively.

For every atomic step $\pi \in \Pi$, the set $V_f(\pi)$ contains all free variables of the atomic step, the set $V_d(\pi)$ contains all variables declared by the atomic step inside the action context, and the set $V_a(\pi)$ contains the subset of the assignable variables of the actor context and the action context that are assigned to a new value by the atomic step.

Furthermore, we define for every atomic step $\pi \in (\Pi_{IG} \cup \Pi_I \cup \Pi_{OG} \cup \Pi_O)$ a function $port(\pi)$, which returns the *port tag* of the port, the atomic step is related to, and for every atomic step $\pi \in \Pi_S$ a function $order(\pi)$, which returns the lexical position of the CAL statement the atomic step is related to, such that if a statement a appears lexically before b in the CAL code of an actor, then $order(\pi_a) < order(\pi_b)$.

The process of transforming an action from CAL to Calflow can now be broken down into four steps:

- *transform* every component of A into one or more atomic steps of Π,

- *analyze* every atomic step $\pi \in \Pi$ to generate $V_f(\pi)$, $V_d(\pi)$ and $V_a(\pi)$,

- *generate* the set Δ of dependencies between the atomic steps in Π,

- *create* an empty action schedule τ.

Transforming CAL Action Components into Calflow Atomic Steps

All CAL action components are transformed according to the following rules:

- $i \in I \Rightarrow \pi_0 \in \Pi_{IG},\ \pi_1 \in \Pi_I$: every *input pattern* is transformed into one *input guard atomic step* and one *input atomic step*.

- $o \in O \Rightarrow \pi_0 \in \Pi_{OG},\ \pi_1 \in \Pi_O$: every *output pattern* is transformed into one *output guard atomic step* and one *output atomic step*.

- $g \in G \Rightarrow \pi_0 \in \Pi_G$: every *guard* is transformed into one *guard atomic step*.

- $d \in D \Rightarrow \pi_0 \in \Pi_D$: every *variable declaration* is transformed into one *decl atomic step*.

- $s \in S \Rightarrow \pi_0 \in \Pi_S$: every *statement* is transformed into one *stmt atomic step*.

Analyzing Atomic Steps

After transformation, all atomic steps are analysed and the following sets are created:

- $V_f(\pi) = V_{free}(\pi) \cup V_{lazy}(\pi)$, where $V_{free}(\pi)$ contains all free variables that occur outside of closures in any expression or statement of the atomic step and $V_{lazy}(\pi)$ contains the free variables that occur inside a closure in any expression of the atomic step. The latter are called *lazy variables*[1].

- $V_d(\pi)$ contains all variables declared by the atomic step inside the action context. Except for *Input* and *Decl* atomic steps, the set is always empty.

- $V_a(\pi)$ contains the subset of the assignable variables of the action context and the action that are assigned to a new value in any statement of the atomic step. Except for *Stmt* atomic steps, the set is always empty.

---

[1] A variable that occurs free inside a closure does not need to be declared when the closure is declared, but when the closure is used for the first time. The closure declaration is said to depend *lazy* from the variable declaration. See [9] for more details.

Generating Dependencies

A dependency $\delta \in \Delta$ is a tuple $(\pi_\alpha, \pi_\beta) \in \Pi^2$ and states that the atomic step π_β depends on the atomic step π_α.

The non-lazy dependency set Δ_{nl} is the set of all tuples $(\pi_\alpha, \pi_\beta) \in \Pi^2$, for which one of the following predicates is true:

- $\pi_\alpha \in \Pi_{IG} \wedge \pi_\beta \in \Pi_I \wedge port(\pi_\alpha) = port(\pi_\beta)$

- $\pi_\alpha \in \Pi_{OG} \wedge \pi_\beta \in \Pi_O \wedge port(\pi_\alpha) = port(\pi_\beta)$

- $\pi_\alpha, \pi_\beta \in (\Pi_{IG} \cup \Pi_I \cup \Pi_{OG} \cup \Pi_O \cup \Pi_G \cup \Pi_D) \wedge V_d(\pi_\alpha) \cap V_{free}(\pi_\beta) \neq \varnothing$

- $\pi_\alpha \in (\Pi_I, \Pi_D) \wedge \pi_\beta \in \Pi_S \wedge V_d(\pi_\alpha) \cap V_f(\pi_\beta) \cup V_d(\pi_\alpha) \cap V_a(\pi_\beta) \neq \varnothing$

- $\pi_\alpha, \pi_\beta \in \Pi_S$
  $\wedge V_a(\pi_\alpha) \cap V_f(\pi_\beta) \cup V_a(\pi_\alpha) \cap V_a(\pi_\beta) \cup V_f(\pi_\alpha) \cap V_a(\pi_\beta) \neq \varnothing$
  $\wedge order(\pi_\alpha) < order(\pi_\beta)$

- $\pi_\alpha \in \Pi_S \wedge \pi_\beta \in \Pi_O \wedge V_a(\pi_\alpha) \cap V_f(\pi_\beta) \neq \varnothing$

- $\pi_\alpha \in (\Pi_{IG}, \Pi_{OG}, \Pi_G) \wedge \pi_\beta \in (\Pi_O, \Pi_S)$

Dependencies, for which one of the first six predicates are true, are called *data dependencies* and dependencies, for which only the last predicate is true, are called *constraint dependencies*.

The lazy dependency set Δ_{lazy} is the set of all tuples $(\pi_\alpha, \pi_\beta) \in \Pi^2$, for which the following predicate is true:

- $\pi_\alpha, \pi_\beta \in (\Pi_{IG} \cup \Pi_I \cup \Pi_{OG} \cup \Pi_O \cup \Pi_G \cup \Pi_D) \wedge V_d(\pi_\alpha) \cap V_{lazy}(\pi_\beta) \neq \varnothing$

Using the non-lazy and the lazy dependency set, the dependency set Δ of the Calfow action *A'* is the union of Δ_{nl} and the transitive hull of Δ_{lazy} in Δ_{nl} [2].

---

[2] In the implementation of the transformation, a more complex algorithm is used to reduce the needed number of dependencies.

A.2 Action Scheduling

Every action of a Calflow actor may optionally contain an *action schedule*, which defines a partial or total order on all or some atomic steps of the action. Generating such an action schedule will usually be one of the first steps of code generation, to define the execution order of the atomic steps.

With adherence of the dependencies of an action and depending on the area of interest, different action schedules may be generated which aim to optimize different goals.

A.2.1 A Simple Sequential Action Schedule

A simple scheduler was implemented for this thesis, which generates a sequential action schedule of the atomic steps in an action. Even though any valid action schedule would have been fine for our purposes, the implemented action schedule tries to generate a schedule which could also be used for real implementations.

The complexity of finding an optimal schedule is usually NP, therefore the presented scheduler uses a heuristics, which tries to optimize the declared goals as good as possible. The scheduler could still be trimmed to generate in average better solutions, however this was not the goal of this thesis.

Optimization Goals
In the following list, the optimization goals are presented with decreasing priority from top to down. The presented schedule tries to optimize the following goals:

- as long as action matching is not finished, try to avoid consuming tokens as long as possible,

- evaluate action matching as fast as possible,

- when action matching is finished, try to generate output tokens as fast as possible,

- when all output tokens are generated, simply finish without priorities.

Algorithm
Lets define a function $path(\pi_\alpha, \pi_\beta)$, which returns the shortest path $p = (\pi_0, \pi_1, ..., \pi_n)$ from $\pi_0 = \pi_\alpha$ to $\pi_n = \pi_\beta$, such that $\forall i \in [0, n-1]$: $\pi_i, \pi_{i+1} \in \Pi$ and $(\pi_i, \pi_{i+1}) \in \Delta_D$ or $p = (\pi_\alpha)$ if $\pi_\alpha = \pi_\beta$. If no path exists at all,

the function returns $p = (\ )$. We define the *length* of p as the number of atomic steps in the path minus one, and as not applicable if p is empty.

Further, for every atomic step $\pi \in \Pi$,

- $d_G(\pi) = \min\limits_{\pi_i \in \Pi_G} (path(\pi,\ \pi_i))$ is the length of the shortest path to a guard atomic step,

- $i_G(\pi)$ equals the number of input atomic steps in the shortest path to a guard atomic step,

- $d_O(\pi) = \min\limits_{\pi_i \in \Pi_O} (path(\pi,\ \pi_i))$ is the length of the shortest path to an output atomic step,

- $\Delta(\pi)$ is the dependency set of π and contains all dependencies $(\pi_i,\ \pi) \in \Delta$ with $\pi_i \in \Pi$.

Using these definitions, we can describe the used algorithm:

Simple_Sequential_Scheduler {

$\quad \Pi_{NS} = \Pi$*;*

$\quad$*while(*$\Pi_{NS} \neq \varnothing$*) {*

$\quad\quad \Pi_A \subseteq \Pi_{NS}$*, such that* $\forall\, \pi \in \Pi_A \rightarrow \Delta(\pi) = \varnothing$*;*

$\quad\quad \Pi_{dG} \subseteq \Pi_A$*, such that* $\forall\, \pi \in \Pi_{dG} \rightarrow d_G(\pi) \geq 0$*;*

$\quad\quad \Pi_{dG}{}^* \subseteq \Pi_{dG}$*, such that* $\forall\, \pi \in \Pi_{dG}{}^* \rightarrow i_G(\pi) = \min\limits_{\forall\, \pi_j \in \Pi_{dG}} (i_G(\pi_j))$*;*

$\quad\quad \Pi_{dO} \subseteq \Pi_A$*, such that* $\forall\, \pi \in \Pi_{dO} \rightarrow d_O(\pi) \geq 0$*;*

$\quad\quad \Pi_{dO}{}^* \subseteq \Pi_{dO}$*, such that* $\forall\, \pi \in \Pi_{dO}{}^* \rightarrow d_O(\pi) = \min\limits_{\forall\, \pi_j \in \Pi_{dO}} (d_O(\pi_j))$*;*

$\quad\quad$*if (*$\Pi_{dG}{}^* \neq \varnothing$*) {*

$\quad\quad\quad$*pick* $\pi' \in \Pi_{dG}{}^*$*;*

$\quad\quad$*} else if (*$\Pi_{dO}{}^* \neq \varnothing$*) {*

$\quad\quad\quad$*pick* $\pi' \in \Pi_{dO}{}^*$*;*

$\quad\quad$*} else {*

$\quad\quad\quad$*pick* $\pi' \in \Pi_A$*;*

$\quad\quad$*}*

```
            schedule π' ;
            Π_NS = Π_NS\π' ;
            ∀ π ∈ Π_NS → Δ(π) = Δ(π)\π' ;
        }
    }
```

A.3 A Complex Example of a Calflow Actor

All actor examples presented in this report have fairly simple data dependencies. This chapter however, presents a CAL actor with very complex data dependencies. In fact, this example was used as test case during the development of the CAL to Calflow transformer.

Note, that the action in the presented actor does not perform any useful task.

```
actor TestActor (Integer param_1)
      Integer input_1, Integer input_2, multi Integer input_3,
      multi Integer input_4, Integer input_5 ==> Integer output_1 :

   Integer state := param_1;
   [Integer --> Integer] f
         := lambda (Integer a) --> Integer :
      a
   end;
   [Integer, Integer, Integer --> Integer] g
         := lambda (Integer a, Integer b, Integer c) --> Integer :
      a
   end;
   action
      [in1], [in2] repeat funcA(in5a), [in3] at in1,
      [in4] at x repeat funcB(in5a), [in5a,in5b]
      ==> [out] repeat funcA(in5a)
      guard g(in1, in3, x), in1 != state;
      var Integer out = state,
         Integer x = in5 + 1,
         Integer y = in1 + in3 + x,
         [Integer --> Integer] funcA
               := lambda (Integer a) --> Integer :
            funcA(y)
         end,
         [Integer --> Integer] funcB
               := lambda (Integer a) --> Integer :
            funcA(y)
         end;
      do
         state := state * (1 + in1) + in2;
      end
   end
end
```
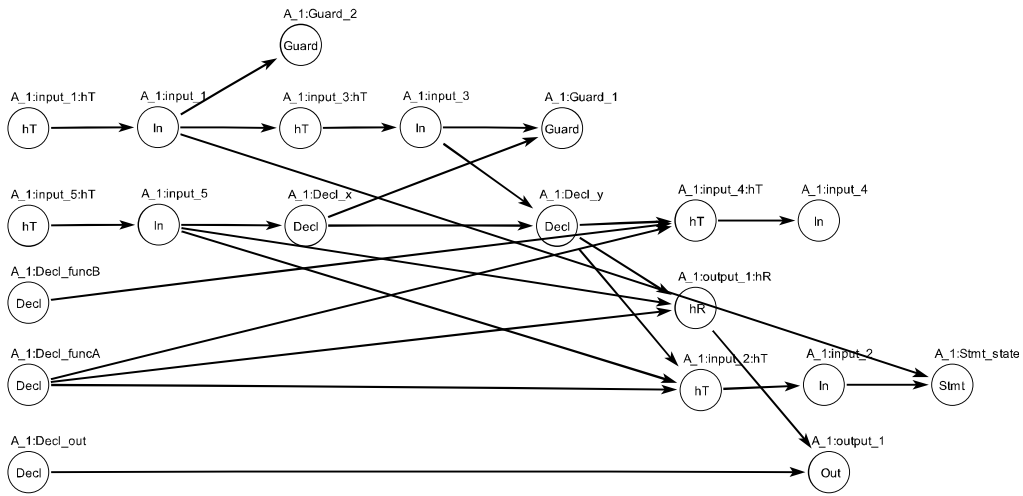
Fig. A-1:   The dataflow graph representation of the action in the TestActor.
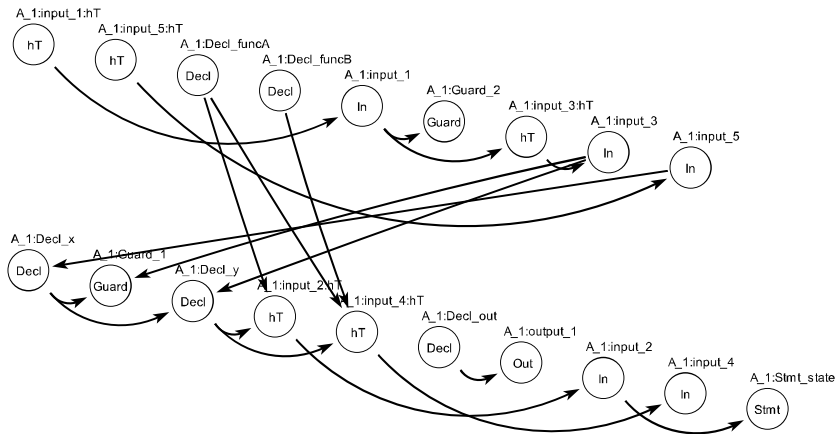
Fig. A-2:   The atomic steps of the action in the TestActor with a total order, defined by the action schedule presented in this chapter.

Appendix B

Software

This appendix presents an overview of the software, which was developed during the work on this thesis.

B.1 Environment

We use XML as file formats for Cal, Calflow, CIA and the description of models. All file formats are defined in XML Schema [17]. The transformations are implemented using XSLT [18], and Xchain is used as command line tool to manage and invoke them.

B.2 Implemented Transformations

Figure B-1 shows the different file formats and the transformations available for each format. The Ptolemy MoML file can be opened in Ptolemy II and provides a graphical representation of Calflow actions or CIAs[1]. All transformations are invoked using the following command:

```
java caltrop.Xchain <command> <sourcefiles>
```

Some more commands, which provide shortcuts to transform for example CalML directly into CiaML, are available and are documented using the following command:

```
java caltrop.Xchain -doc
```

---

[1] The graphical representations do not contain any Ptolemy II semantics.

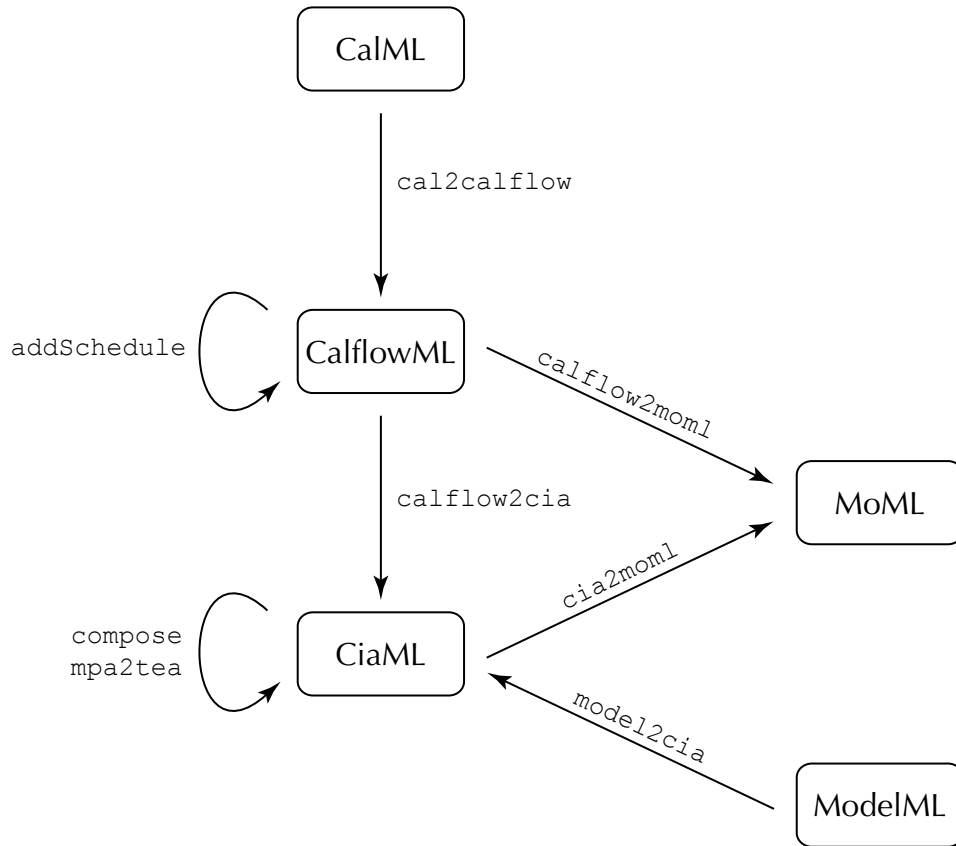**Fig. B-1:**   File formats and available transformation commands.

Additionally, the following command takes a ModelML file and creates the CIAs of all actors, the CIA of the model, the MPA, the TEA and a Ptolemy MoML file for graphical representation of all generated files:

```
java caltrop.cia.CreateModel <modelfile>
```

Xchain –
A Framework for XML Processing

Xchain is a framework for XML processing which was developed for the Caltrop project, as part of this thesis.

The Caltrop project makes extensive use of XSLT as a programming language to implement complex program transformations, program compositions and code generation. To handle the complexity of these tasks it is often appropriate to split a task into a number of smaller tasks, each one implemented as an own XSLT program which generates an intermediate result. A complex transformation is then achieved by consecutive application of each of the smaller transformations.

By using this approach one ends up with a number of relatively small and therefore easy to understand and easy to maintain XSLT programs.

The problem then arises how to specify and maintain the chains of XSLT programs and how to apply them to a XML document in an simple, fast and user friendly way.

The aim of the Xchain project is to provide a solution to the above mentioned problems. Xchain even takes the approach one step further by not only providing a solution to specify chains of XSLT programs, but by introducing a completely dataflow based concept of XML processing with filters blocks. In Xchain every step of XML processing, as for example parsing of a source document, serialization of a DOM-tree or application of an XSLT transformation, is handled in an own filter block and a complete workflow process is defined as a consecutive application of a number of different filter blocks, specified as a filter chain. In this context, these filter chains are called Xchains and give the name to the software package.

The Xchain package defines an XML Schema to specify filter chains and to configure the filter blocks contained in a chain, it also defines and implements a Java API for using Xchain in Java applications and furthermore it comes with a Java based command line tool for using Xchain manually.

C.1 The Concept of Filter Based Processing in Xchain

In Xchain every step of XML processing is done by applying an appropriate filter to an XML document. More precisely a filter in Xchain works on the DOM representation of an XML document. Figure C-1 shows a typical filter block. It takes a DOM tree as input, executes and then returns a DOM tree on its output.

Fig. C-1:   A filter for DOM tree processing in Xchain.

C.2 Xchains

To do something useful with an XML document, usually a chain of filters needs to be applied to a document. Such a chain may itself be represented as a single filter and in fact a filter type, named Xchain filter, is defined which simply contains such a chain of other filters. When a Xchain filter is applied, it feeds its input into the first filter of its internal filter chain, the result of this first filter is fed into the second filter, and so on until all filters in the internal chain are applied. Finally, the result of the last internal filter is returned as output of the Xchain filter.

Figure C-2 shows an example of a Xchain filter. The Xchain in this example opens a XML document, applies a number of XSLT transformations to it, saves an intermediate result and then applies some more XSLT transformations before it finally saves the final result.

Detailed information about all the different filter types in this example and more can be found in Chapter C.6.
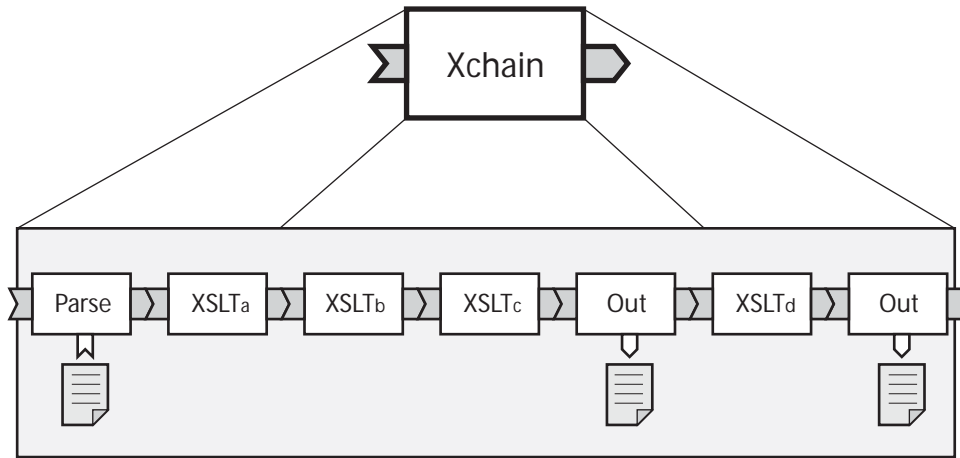
Fig. C-2:   An example of a Xchain filter containing a filter chain.

C.3 The Filter Context

In the example mentioned above, the Parse filter parses a XML document to generate a DOM-tree. The information needed to find the correct file to parse, i.e. the path-name of the XML source file, may be given to the filter with its configuration data at its creation time. However it would be more useful if the filter could access this information at run-time. This would for example allow to apply the given Xchain filter to a number of different XML source documents without creating a new instance of the filter for each document.

For this purpose all filters in Xchain are embedded in a so called filter context. The filter context possesses a number of properties and contains a set of Xchain filters. Further the filter context provides methods for its embedded filters to access its properties and it provides a central place for the filters to store and read arbitrary variables.

Using the filter context in the above example, the path of the XML source document could be contained in a property which the Parse filter would read each time it is invoked. A user could then simply update this property each time before invoking the Parse filter.

C.4 XchainML

XchainML is a XML format, defined in a XML Schema, which can be used to specify Xchain filters and to define and configure the filter blocks contained in them.

Xchain is usually used together with a configuration file written in XchainML. This configuration file is used by the filter context to set up all needed filter chains and properties.

XML Schema (excerpt)

```
<xs:element name="project">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="property" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:attribute name="name" type="xs:string"
                     use="required"/>
               <xs:attribute name="value" type="xs:string"
                     use="optional"/>
            </xs:complexType>
         </xs:element>
         <xs:element ref="xchain" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
</xs:element>
```

C.5 The Implemented Filter Context

Xchain v1.0 comes with one implemented filter context, namely Xchain.java. This implementation can be used both as a stand alone command line tool as well as in another Java program by creating an instance of it.

The implementation provides a number of different `executeCommand()` methods which can be used to execute any named Xchain filter which is defined as top-level element in the used XchainML configuration file.

Most of the times, Xchain is used to parse a source-file, apply a number of transformations on it and save the result in a new file. To support this use-case, one of the provided `executeCommand()` methods takes a source-file URI as an additional argument. The passed string in this argument is used to set up a number of properties before actually invoking the specified Xchain filter. The Parse and Out filters in the invoked Xchain filter may then read these properties to locate the source-file or to generate the names of result files.

If, for example, the value of the passed source-file URI argument is "`docs/exam-ples/Test.calml.xml`", the following properties are set up by the filter context:

```
CURRENT_SOURCE_FILE_URI=docs/examples/Test.calml.xml
CURRENT_SOURCE_FILE_DIR=docs/examples
CURRENT_SOURCE_FILE_FULL_NAME=Test.calml.xml
CURRENT_SOURCE_FILE_NAME=Test
CURRENT_SOURCE_FILE_EXT=.calml.xml
```

C.6 The Implemented Filters

Xchain v1.0 comes with ten implemented filters. Table C-1 gives a short overview and the following chapters provide detailed informations on each filter.

Table C-1:   An overview of all filters in Xchain v1.0

Name	Short Description
Xchain Filter	Defines and contains a chain of filters.
XchainRef Filter	References to a named top-level Xchain filter.
Branch Filter	Branches a filter chain.
Parse Filter	Parses a source file.
Out Filter	Serializes a DOM-tree and saves the result in a file.
XSL Filter	Applies a XSLT transformation.
Save Filter	Saves a DOM-tree in a variable of the filter context.
Load Filter	Loads a DOM-tree from a variable of the filter context.
Call Filter	Calls an external Java method.
Msg Filter	Writes a message to the system standard output.

C.6.1 General Implementation Details

Some implementation details apply to all implemented filters and are described here.

Configuration

XchainML configuration files are used to setup Xchain. In a configuration file, the configuration data of every filter is contained in a single XML element. When a filter is instantiated, this element is passed to a filter factory which, based on the tag name of the element, finds the correct implementation class for the filter. The element is then passed on to the appropriate constructor of the filter class which creates an instance of the filter configured with the configuration data contained in the element.

A filter does not provide any accessor methods to change its configuration after instanciation. The reason for this is that for many filter types, changing the configuration after instanciation would lead to a low performance. Instead, the property resolving mechanism described below provides a way how filters may be implemented that want to provide the possibility to change configuration. The difference to accessor methods is that with the property resolving mechanism a filter may or may not pull new configuration data from its filter context rather than the data being pushed into it.

In the attribute tables, which are provided for every filter, the last column is titled PLU, which stands for "Property Lookup". If the value in this column is "Init", this means that the filter makes only a property lookup for the given attribute at instantiation time. If on the other hand the value in this column is "RT", this means that the filter makes a property lookup for the given attribute at runtime on each invocation.

Execution Mode

Every filter has an attribute named "mode" with an arbitrary string value. Additionally the filter context has one property which is also named "mode".

When a filter is executed it first checks the value of its mode attribute. If the value is not an empty string it looks up the mode property of the filter context and compares it to the value of its own mode attribute. If the two values are not equal the filter does not execute and simply returns the input it got without any change.

Different execution modes may be used for example for debugging, where a number of Out filters with a mode attribute set to "debug" could be scattered along a chain of XSL filters. When the mode property of the filter context is then set to "debug", these Out filters would produce a number of intermediate results for debugging, while in normal execution mode they would simply do nothing.

Property Resolving

As mentioned above, filters may access the properties of the filter context. Filters may for example change their configuration by accessing these properties or they may use these properties to pass their values as parameters to an XSLT program.

All filters allow their attributes to contain placeholders for properties which are replaced with the value of the corresponding property in the filter context. All of these placeholders are resolved at instantiation time and some get resolved each time the filter is invoked.

A property placeholder consists of the property name enclosed by curly brackets and a leading dollar sign: `${propertyName}`

For example consider the following attribute value:

```
"${modelName}_${directorName}.${fileType}.xml"
```

And suppose the filter context contains the following properties:

```
modelName=SourceActor
directorName=SDF
fileType=cia
```

Then the attribute value defined above would be resolved at runtime to:

```
"SourceActor_SDF.cia.xml"
```
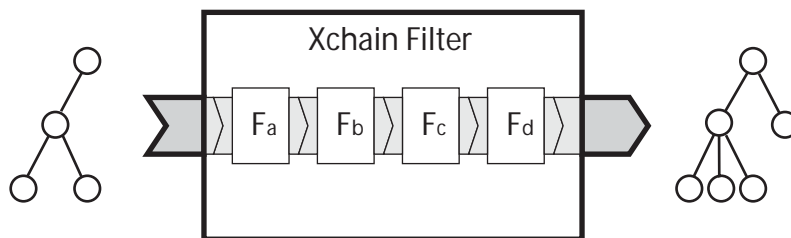
C.6.2 Xchain Filter

Fig. C-3:   Dataflow in a Xchain Filter.

The Xchain filter contains a filter chain which is fed with the input DOM-tree. The result returned from the last filter in the filter chain is passed to the filter output.

Attributes

Name	Description	PLU
`name`	The name which can be used to identify the Xchain filter.	Init
`type`	The access type of the Xchain filter, either `public` or `private`.	Init
`mode`	The execution mode of the filter.	RT

Child Elements

Name	Description
`<doc>`	This element may contain a short documentation of what the filter does.
`<#any_filter#>`	Any number of elements describing the filter chain contained in the Xchain filter. The order of the filters in the chain equals the order of the description elements.

XML Schema

```
<xs:element name="xchain">
   <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
         <xs:element name="doc" type="xs:string" minOccurs="0"/>
         <xs:choice>
            <xs:element ref="call" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="load" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="msg" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="out" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="parse" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="save" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="xchain" minOccurs="0"
                  maxOccurs="unbounded"/>
            <xs:element ref="xchainref" minOccurs="0"
                  maxOccurs="unbounded"/>
            <xs:element ref="xsl" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="branch" minOccurs="0"
                  maxOccurs="unbounded"/>
         </xs:choice>
      </xs:sequence>
      <xs:attribute name="name" type="Identifier" use="optional"/>
      <xs:attribute name="type" type="XchainType" use="optional"/>
      <xs:attribute name="mode" type="xs:string" use="optional"/>
   </xs:complexType>
</xs:element>
```

Example

```
<xchain name="ExampleXchain" type="public">
    <doc>This is an example.</doc>
    <msg>Transforming ${CURRENT_SOURCE_FILE_URI} ...</msg>
    <parse href="${CURRENT_SOURCE_FILE_URI}"/>
    <xsl href="xsl/XSLTa.xsl"/>
    <xsl href="xsl/XSLTb.xsl"/>
    <xsl href="xsl/XSLTc.xsl"/>
    <out method="xml" indent="yes"
         dir="."
         filename="${CURRENT_SOURCE_FILE_NAME}"
         fileext=".step1.xml"/>
    <xsl href="xsl/XSLTd.xsl"/>
    <out method="xml" indent="yes"
         dir="."
         filename="${CURRENT_SOURCE_FILE_NAME}"
         fileext=".step2.xml"/>
</xchain>
```

C.6.3 XchainRef Filter

The XchainRef filter is used to reference to a named Xchain filter which is defined as top-level element in a XchainML configuration file.

Attributes

Name	Description	PLU
name	The name of the Xchain filter which is referenced.	Init
mode	The execution mode of the filter.	RT

Child Elements

XML Schema

```
<xs:element name="xchainref">
    <xs:complexType>
        <xs:attribute name="name" type="Identifier" use="required"/>
        <xs:attribute name="mode" type="xs:string" use="optional"/>
    </xs:complexType>
</xs:element>
```

Example

```
<xchainref name="ExampleXchain"/>
```
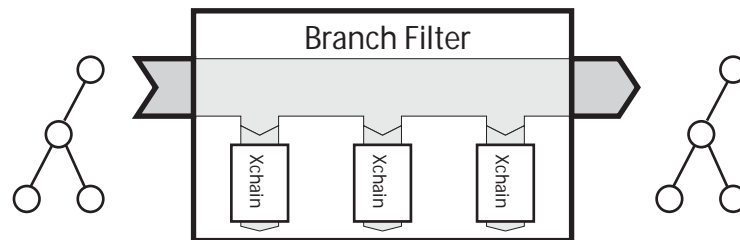
C.6.4 Branch Filter

Fig. C-4:   Dataflow in a Branch Filter

The Branch filter provides a way to branch the processing flow of a filter chain. A Branch Filter may contain any number of Xchain filters or Xchainref filters. When the Branch filter is invoked, it applies each of its internal Xchain and Xchainref filters to the input DOM-tree and finally returns the input DOM-tree without any change on its output.

The outputs of all the internal Xchain and Xchainref filters get discarded and are not accessible from anywhere. However usually each of the internal Xchain and Xchainref filters would contain an Out filter at the end of their filter chain, which would save the resulting DOM-tree in a document.

Attributes

Name	Description	PLU
`mode`	The execution mode of the filter.	RT

Child Elements

Name	Description
`<xchain>`	A definition of a new Xchain filter that should be branch to.
`<xchainref>`	A reference to an existing Xchain filter that should be branched to.

XML Schema

```
<xs:element name="branch">
   <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
         <xs:element ref="xchain" minOccurs="0" maxOccurs="unbounded"/>
         <xs:element ref="xchainref" minOccurs="0"
               maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="mode" type="xs:string" use="optional"/>
   </xs:complexType>
</xs:element>
```

Example

```
<branch>
   <xchain>
      <xchainref name="_FooXchain"/>
      <out method="xml" indent="yes"
            dir="."
            filename="${CURRENT_SOURCE_FILE_NAME}"
            fileext=".foo.xml"/>
   </xchain>
   <xchain>
      <xchainref name="_BarXchain"/>
      <out method="xml" indent="yes"
            dir="."
            filename="${CURRENT_SOURCE_FILE_NAME}"
            fileext=".bar.xml"/>
   </xchain>
</branch>
```
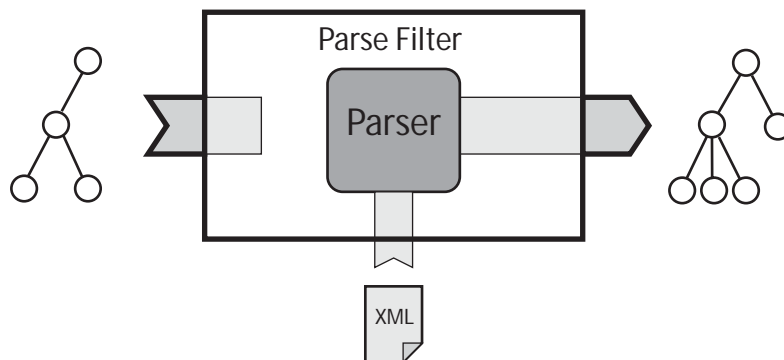
C.6.5 Parse Filter

Fig. C-5:   Dataflow in a Parse Filter

The Parse filter parses a XML source document and generates a DOM-tree of it which it then returns on its output. The DOM-tree arriving on the input is discarded.

Attributes

Name	Description	PLU
href	The URI of the source document.	RT
documentbuilder	The Document Builder implementation that should be used for parsing the document. If this attribute is not present, the default value of the filter context, defined in a property named defaultDocumentBuilderFactory, will be used.	Init
namespaceAware	Specifies whether the parser should be name-space aware or not.	Init
validate	Specifies whether the parser should validate the document or not.	Init
mode	The execution mode of the filter.	RT

Child Elements

XML Schema

```
<xs:element name="parse">
   <xs:complexType>
      <xs:attribute name="href" type="xs:string" use="required"/>
      <xs:attribute name="documentbuilder"
            type="xs:string" use="optional"/>
      <xs:attribute name="namespaceAware"
            type="xs:string" use="optional"/>
      <xs:attribute name="validate" type="xs:string" use="optional"/>
      <xs:attribute name="mode" type="xs:string" use="optional"/>
   </xs:complexType>
</xs:element>
```

Example

```
<parse href="${CURRENT_SOURCE_FILE_URI}"
      documentbuilder="com.icl.saxon.om.DocumentBuilderFactoryImpl"
      namespaceAware="false" validate="false"/>
```
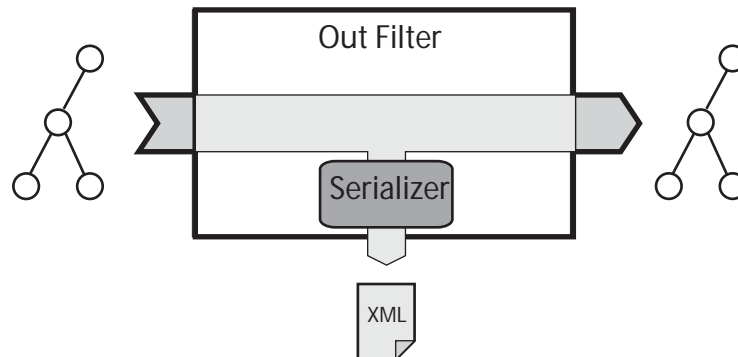
C.6.6 Out Filter

Fig. C-6:   Dataflow in a Out Filter

The Out filter serializes its input DOM-tree and saves it in a document. The input DOM-tree is then returned at the output without any change.

The Out filter provides three attributes, namely `dir`, `filename` and `fileext`, to specify the location where the document should be stored. At runtime the location information is created by concatenating the values of the three attributes and by adding a path separator between `dir` and `filename`. Therefore the value of the `dir` attribute should not end with a path separator.

Attributes

Name	Description	PLU
method	The output mode that should be used to serialize the DOM-tree.	Init
dir	The path of the directory where the output document should be stored. Both "/" and "\" are allowed as separators and get automatically replaced by the system dependent separator at runtime.	RT
filename	The name of the output document.	RT
fileext	The file extension of the output document.	RT

Name	Description	PLU
transformer	The XSLT Transformer implementation that should be used to serialize the DOM-tree. If this attribute is not present, the default value of the filter context will be used.	Init
mode	The execution mode of the filter	RT
##any	Any other attribute will be passed directly to the XSLT Transformer for configuration.	Init

Child Elements

XML Schema

```
<xs:element name="out">
   <xs:complexType>
      <xs:attribute name="method" type="xs:string" use="optional"/>
      <xs:attribute name="dir" type="xs:string" use="required"/>
      <xs:attribute name="filename" type="xs:string" use="required"/>
      <xs:attribute name="fileext" type="xs:string" use="required"/>
      <xs:attribute name="transfomer" type="xs:string" use="optional"/>
      <xs:attribute name="mode" type="xs:string" use="optional"/>
      <xs:anyAttribute namespace="##any"/>
   </xs:complexType>
</xs:element>
```

Example

```
<out method="xml" indent="yes"
     dir="result/debug"
     filename="${CURRENT_SOURCE_FILE_NAME}"
     fileext=".debug1.xml"
     mode="debug"/>
```

C.6.7 XSL Filter
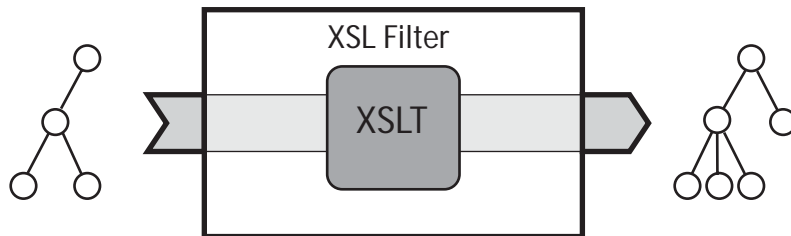
Fig. C-7:   Dataflow in a XSL Filter

The XSL filter applies a XSLT transformation to the input DOM-tree and returns the result of the transformation on its output.

Attributes

Name	Description	PLU
`href`	The URI of the source document containing the XSLT program.	Init
`transformer`	The XSLT Transformer implementation that should be used to serialize the DOM-tree. If this attribute is not present, the default value of the filter context, defined in a property named `defaultTransformerFactory`, will be used.	Init
`mode`	The execution mode of the filter	RT

Child Elements

Name	Description
`<param>`	Defines parameters that should be passed to the XSLT program.

XML Schema

```
<xs:element name="xsl">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="param" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:attribute name="name"
                       type="xs:string" use="required"/>
               <xs:attribute name="value"
                       type="xs:string" use="optional"/>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
      <xs:attribute name="href" type="xs:anyURI" use="required"/>
      <xs:attribute name="transformer" type="xs:string" use="optional"/>
      <xs:attribute name="mode" type="xs:string" use="optional"/>
   </xs:complexType>
</xs:element>
```

Example

```
<xsl href="xsl/foo.xsl"
     transformer="com.icl.saxon.TransformerFactoryImpl">
   <param name="param1" value="true"/>
   <param name="param2" value="15"/>
</xsl>
```
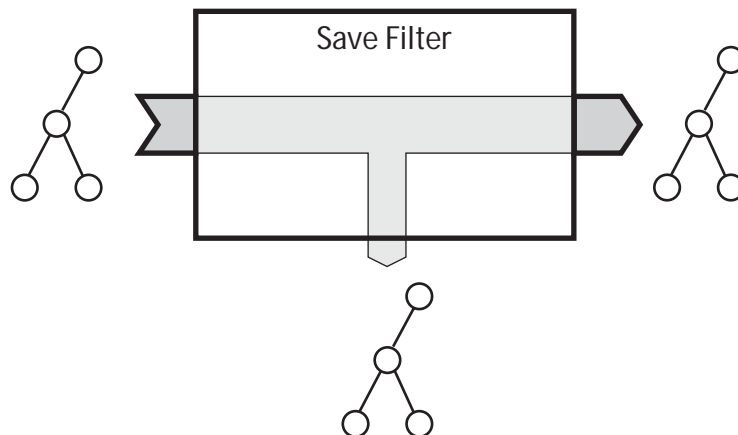
C.6.8 Save Filter

Fig. C-8:   Dataflow in a Save Filter

The Save filter saves the input DOM-tree in a variable in the filter context. The saved DOM-tree can later be loaded again from the filter context by using a Load filter.

Attributes

Name	Description	PLU
name	The name of the variable in which the input DOM-tree will be stored in the filter context.	RT
mode	The execution mode of the filter	RT

Child Elements

XML Schema

```
<xs:element name="save">
   <xs:complexType>
      <xs:attribute name="name" type="Identifier" use="required"/>
      <xs:attribute name="mode" type="xs:string" use="optional"/>
   </xs:complexType>
</xs:element>
```

Example

```
<save name="temp"/>
```
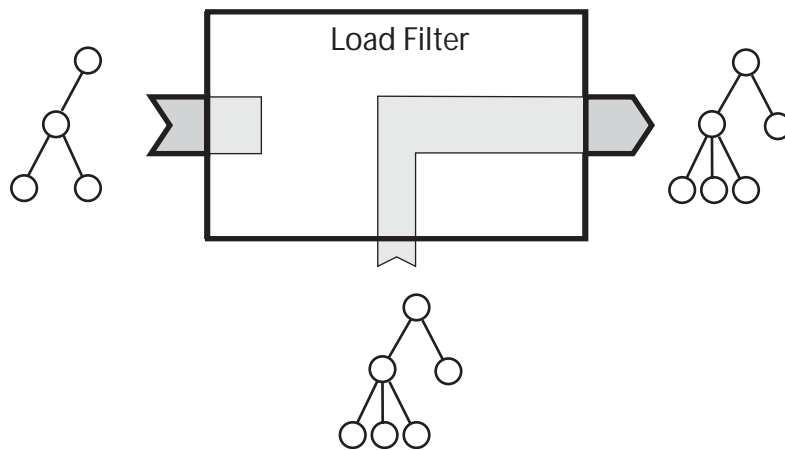
C.6.9 Load Filter

Fig. C-9:   Dataflow in a Load Filter

The Load filter loads a DOM-tree that was previously saved in a variable of the filter context by a Save filter.

Attributes

Name	Description	PLU
name	The name of the variable in the filter context that contains the desired DOM-tree.	RT
mode	The execution mode of the filter	RT

Child Elements

XML Schema

```
<xs:element name="load">
   <xs:complexType>
      <xs:attribute name="name" type="Identifier" use="required"/>
      <xs:attribute name="mode" type="xs:string" use="optional"/>
   </xs:complexType>
</xs:element>
```

Example

```
<load name="temp"/>
```

C.6.10 Call Filter
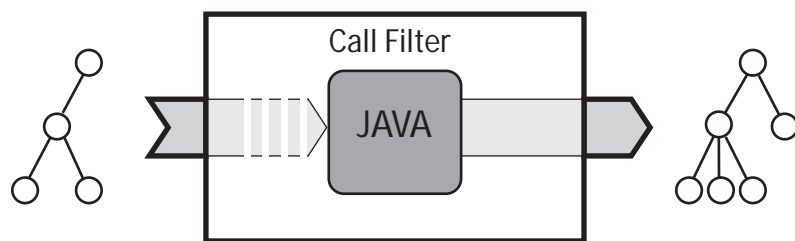
Fig. C-10:   Dataflow in a Call Filter

The call filter is used to call an external Java method. It can invoke any static or non-static method with a return type of either `org.w3c.dom.Document`, `java.lang.String` or `void`.

If the invoked method is not static, the information in the `<init>` element is used to create an instance of the class, otherwise the `<init>` element is ignored. The information in the `<method>` element is then used to find the correct method and to check

its return type. If the return type is `org.w3c.dom.Document`, the result of the method call is directly passed on to the filter output. If the return type is `java.lang.String`, the result of the method call is first parsed using the parser configuration given in the <result> element, then the resulting DOM-tree is passed on to the filter output. And finally, if the return type is void, the filter assumes that the method will write to the system standard output and therefore reads the system standard output into a string and parses it in the same way as it would do when the return type would be directly of type `java.lang.String`.

In the `<init>` and `<method>` elements, the arguments of the constructor and the method call can be specified. For this purpose both elements may contain any number of `<arg>` child elements.

Every `<arg>` element has two attributes, namely type and value, which contain the information to create an object of any class that provides a string constructor, which is used as argument for the constructor or the method call.

Usually the value is interpreted as a `java.lang.String` and is used to invoke the string constructor of the specified class to generate an instance of it. The following three values however are interpreted in a special way by the filter:

Value	Description
${NULL}	This value is replaced with a Java `null`.
${FILTER_INPUT}	This value is replace with the input DOM-tree of the filter.
${SERIALIZED_FILTER_INPUT}	This value is replaced with a serialized version of the input DOM-tree of the filter.

It is also possible to generate arrays of objects as arguments. For this, the type attribute must end with "[]" and the `<arg>` element can contain any number of `<elem>` elements. An array of the specified type is then created and the information of the `<elem>` elements is used to create the elements of the array as well as its length.

Attributes

Name	Description	PLU
class	The name Java class that should be called.	RT
mode	The execution mode of the filter	RT

Child Elements

Name	Description
`<init>`	Contains information on how to instantiate an object of the needed class if the called method is not static.
`<method>`	Contains information on how to call the needed method.
`<result>`	Contains information on how to parse the returned result of the method if it is not a DOM-tree already.

XML Schema

```
<xs:element name="call">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="init" minOccurs="0">
            [(see below)]
         </xs:element>
         <xs:element name="method">
            [(see below)]
         </xs:element>
         <xs:element name="result" minOccurs="0">
            [(see below)]
         </xs:element>
      </xs:sequence>
      <xs:attribute name="class" type="xs:string" use="required"/>
      <xs:attribute name="mode" type="xs:string" use="optional"/>
   </xs:complexType>
</xs:element>
```

```
<xs:element name="init" minOccurs="0">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="arg" minOccurs="0" maxOccurs="unbounded">
            [(see below)]
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

```
<xs:element name="method">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="arg" minOccurs="0" maxOccurs="unbounded">
            [(see below)]
         </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
</xs:element>
```

```
<xs:element name="result" minOccurs="0">
   <xs:complexType>
      <xs:attribute name="documentbuilder" type="xs:string"
            use="optional"/>
      <xs:attribute name="namespaceAware" type="xs:string"
            use="optional"/>
      <xs:attribute name="validate" type="xs:string" use="optional"/>
   </xs:complexType>
</xs:element>
```

```
<xs:element name="arg" minOccurs="0" maxOccurs="unbounded">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="elem" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:attribute name="value" type="xs:string"
                     use="required"/>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
      <xs:attribute name="type" type="xs:string" use="required"/>
      <xs:attribute name="value" type="xs:string" use="required"/>
   </xs:complexType>
</xs:element>
```

Example

```
<call class="caltrop.xchain.FooClass">
   <init>
      <arg type="java.lang.Integer[]">
         <elem value="2"/>
         <elem value="2"/>
         <elem value="1"/>
      </arg>
   </init>
   <method name="bar">
      <arg type="org.w3c.dom.Document" value="${FILTER_INPUT}"/>
      <arg type="java.lang.String" value="empty"/>
   </method>
</call>
```
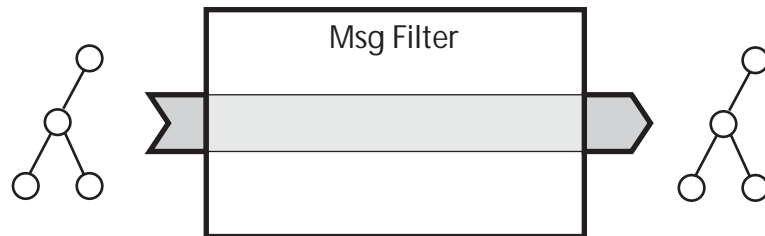
C.6.11 Message Filter

Fig. C-11:   Dataflow in a Msg Filter

The Message filter is used to print a message to the standard system output. It directly passes its input to its output without any change.

Attributes

Name	Description	PLU
mode	The execution mode of the filter	RT

Child Elements

XML Schema

```
<xs:element name="msg">
   <xs:complexType>
      <xs:simpleContent>
         <xs:extension base="xs:string">
            <xs:attribute name="mode" type="xs:string" use="optional"/>
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
</xs:element>
```

Example

```
<msg>Step 1 completed...</msg>
```

C.7 The Xchain Command Line Tool

Xchain comes with a command line tool that can be used to execute filter chains which are defined in an XchainML configuration file. When the command line tool is executed, it looks in the current directory for a file named `xchain.xml` which contains the filter configurations.

The following command is used to invoke Xchain:

```
java caltrop.Xchain <options> <command> <sourcefiles>
```

where `<command>` is the name of the public Xchain filter that should be executed, `<sourcefiles>` are the files that the filter should be applied to and the available `<options>` are:

`-config <configfile>` to specify a different configuration file than xchain.xml,

`-D<name>=<value>` to set a property,

`-doc` to print out the doc-information of all available public Xchain filters,

`-help` to print a help message.

A typical example of an Xchain invokation:

```
java caltrop.Xchain -DhT=true -DhR=false calml2cia *.calml.xml
```

Aufgabenstellung (German)

DA 8, WS 2002/03	Ausgabe: 11.11.2002	IfA Nr. 8911
	Abgabe: 28.03.2003	

Titel der Diplomarbeit	# Statische Prüfung von Aktor Modellen

Autor	Ernesto Wandeler

Beschreibung und Aufgabenstellung	Im Ptolemy Projekt an der UC Berkeley werden die Modellierung, die Simulation und das Design von nebenläufigen, heterogenen Echtzeit-Systemen untersucht. Auf diesem Gebiet der eingebetteten Systeme haben aktororientierte Ansätze in den letzten Jahren zunehmend an Bedeutung gewonnen.

Aktoren sind nebenläufige, parametrisierte Objekte, die zu einer Aktor-Komposition zusammengestellt werden in welcher die einzelnen Aktoren gemäss einem sogenannten Model of Computation durch Austausch von Daten-Token interagieren. Zur Beschreibung von Aktoren wird an der UC Berkeley eine High-Level Aktorsprache namens CAL entwickelt, welche es erlaubt das Verhalten und die Funktionalität von Aktoren plattform-unabhängig zu programmieren.

Das Ziel dieser Diplomarbeit besteht darin, Kompositionen von Aktoren, welche in CAL definiert sind, auf ihre Gültigkeit zu überprüfen.

Bei der Lösung solcher Aufgabenstellungen spielen Automatenmodelle eine wichtige Rolle. Dabei soll zuerst die abstrakte Aktorsprache CAL in geeigneter Weise erweitert werden. Danach soll ein geeignetes Automatenmodell entwickelt werden, welches Aspekte des Verhaltens von Aktoren in geeigneter Weise beschreibt. Im nächsten Schritt soll eine Transformation hergeleitet werden welche aus einem CAL Aktor den entsprechenden Auto-maten extrahiert. Anhand der Komposition der Automaten sollen dann Aussagen über die Aktor-Komposition gemacht werden.

Die Arbeit besteht damit aus einem wesentlichen konzeptionellen Teil sowie auch aus der Implementierung der verschiedenen Konzepte. |
| Zwischenbericht | Am 17.01.2003 ist ein kurzer Zwischenbericht (1-2 Seiten), der sowohl den Stand der Arbeit als auch das geplante Vorgehen beinhaltet, per e-mail an <hagenow@aut.ee.ethz.ch> und an die Betreuer zu schicken. |
| Mündliche Präsentation | Der Termin für die mündliche Präsentation am IfA wird frühzeitig bekannt gegeben. |
| Abgabe des Berichtes | Abgabe des Berichtes am Institut für Automatik, Sekretariat ETL I 23:
- 1 CD und
- 1 weisser, 1-seitiger Originalausdruck (nicht geheftet)

Für das Erstellen des Berichts verweisen wir auf Punkt 2 in den "Vorschriften über die Durchführung von Studien- und Diplomarbeiten". |
| Betreuung | Dr. Joern Janneck/UC Berkeley (Gruppe Prof. Edward Lee)
Prof. W. Schaufelberger |

Arbeitsplatz   UC Berkeley

Fachprofessor ETHZ   Prof. W. Schaufelberger, Institut für Automatik

Bibliography

[1]   The Ptolemy II Project. (*http://ptolemy.eecs.berkeley.edu*)

[2]   The Caltrop Project. (*http://www.gigascale.org/caltrop*)

[3]   Chic – Checker for Interface Compatibility (*http://www-cad.eecs.berkeley.edu/ ~tah/chic/*)

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

[5]   L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, 2001.

[6]   L. de Alfaro and T. A. Henzinger. Interface Theories for Component-Based Design. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, 2001.

[7]   S. S. Bhattacharyya, E. Cheong, J. Davies, M. Goel, B. Kienhuis, C. Hylands, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong and H. Zheng. *Ptolemy II Heterogeneous Concurrent Modeling and Design in Java*. Technical Memorandum UCB/ERL M02/23, University of California, Berkeley, CA 94720, USA, August 2002.

[8]   J. Davies, C. Hylands, J. Janneck, E. A. Lee, J. Liu, S. Neuendorffer, S. Sachs, M. Stewart, K. Vissers, P. Whitaker and Y. Xiong. *Overview of the Ptolemy Project*. Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, CA 94729, USA, March 2001.

[9]   J. Eker and J. W. Janneck. *Caltrop Language Report*. Technical Memorandum UCB/ERL, Electronics Research Lab, Departement of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720, USA, 2002.

[10]   J. Eker and J. W. Janneck. *An Introduction to the Caltrop Actor Language*. Technical Memorandum UCB/ERL, Electronics Research Lab, Departement of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720, USA, 2002.

[11]   C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intellingece*, 8(3):323-363, June 1977.

[12] J. W. Janneck. *Actors and their Composition*. Technical Memorandum UCB/ERL, M02/37, University of California, Berkeley, CA 94720, USA, December 2002.

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

[14] E. A. Lee, S. Neuendorffer and M. J. Wirthlin. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems and Computers*, November 2002.

[15] E. A. Lee and Y. Xiong. *Behavioral Types for Component-Based Design*. Technical Memorandum UCB/ERL, M02/29, University of California, Berkeley, CA 94720, USA, September 2002.

[16] C. A. Petri. *Kommunikation mit Automaten*. Dissertation, Bonn, 1962.

[17] XML Schema. (*http://www.w3.org/XML/Schema*)

[18] XSLT. (*http://www.w3.org/TR/xslt*)