# A Synchronisation Facility for a Stream Processing Coordination Language

Anna Tikhonova

A thesis submitted to the University of Hertfordshire
in partial fulfilment of the requirements of the degree of

Master of Science by Research

15 January 2015

# *Abstract*

In this thesis we present the Astra*Kahn* project that aims to provide environment for stream processing applications with an automatic resource and concurrency management based on communication demand. At the moment the work on the thesis started, the project was at an early stage and there existed no software implementation. The aim of my work is to implement a stream synchronisation facility called *synchroniser* which is a part of the Astra*Kahn* concept. Synchronisers are programmed in a dedicated language. The thesis focuses on the implementation of the language compiler to be integrated into the runtime system prototype being developed in parallel.

Astra*Kahn* structures streaming networks using a fixed set of wiring patterns, including the serial replication. This pattern dynamically replicates its operand network conceptually infinitely many times and wires the replicas in a chain. Astra*Kahn* provides an approach to extract messages from the chain based on the concept of fixed point. The thesis explores the role of synchronisers in forming from a serial replication pipeline.

# Contents

# Chapter 1

# Introduction

The concepts of the new coordination language Astra*Kahn* are described in [1]. The language defines the coordination behaviour of asynchronous stateless components (*boxes*) and their orderly interconnection via stream-carrying channels with finite capacity. Astra*Kahn* structures the interconnect using a fixed set of wiring primitives, viz. serial and parallel composition, wrap-around connection and serial replication. Boxes are connected to the network with one or two input channels and one or more output channels. A stateless box does not synchronise data on its input channels; to this end, Astra*Kahn* provides a synchronisation facility called *synchroniser*. Synchronisers are finite state machines for joining messages and sending them on to the output channels. A synchroniser is connected to the network with one or more input and output channels. Astra*Kahn* provides a dedicated language to define synchronisers, while boxes are specified in any conventional programming language. In order to deal with the issue of application progress, Astra*Kahn* attempts to provide an automatic resource and concurrency management based on communication demand.

## 1.1 Motivation

At the moment the work on this thesis started, the Astra*Kahn* project was at an early stage and there existed no software implementation. In order to carry out research towards automatic resource and concurrency management, an execution environment for Astra*Kahn* applications must be developed. In brief, such an environment includes a compiler to generate an intermediate representation of the application source code and a runtime system to interpret the representation under the input data. Since Astra*Kahn* provides the coordination language and the programming language of synchronisers, the two compilers for them are needed. The grammar of the synchroniser language is given in [1]. The aim of this work is to study synchronisers and implement the synchroniser language compiler.

An Astra*Kahn* component, either a box or a synchroniser, is both a consumer and a producer for some other components in the network. The static correctness of a connection demands that the statically guaranteed properties of an output message be sufficient to satisfy the static requirements of its recipients. In order to check the static correctness over the network, a component can be abstracted with respect to its data-transformation behaviour as a so-called communication passport $p \Rightarrow P$, where $p$ is the conjunction of all the requirements and $P$ is the conjunction of all the guarantees. Since boxes are not analysable by Astra*Kahn* the programmer has to specify passports for them. Synchronisers are fully analysable by Astra*Kahn* and their passports can be extracted from the source code exclusively by program analysis. Such an analysis can be implemented as a part of the synchroniser compiler.

Astra*Kahn* provides the serial replication combinator, which creates a conceptually infinite number of copies of its operand network and connects them in a serial fashion. In Astra*Kahn* the output from the serial replication pipeline is defined using the concept of forward fixed point. A forward fixed point is a message that is not changed by being processed by a newly created replica. In order to detect fixed point messages, the Astra*Kahn* runtime system has to be provided with a pattern that matches all of them. The language report [1] gives an intuitive explanation of how the matching pattern can be embedded into the operand network of the serial replication combinator using synchronisers, so that the programmer does not have to specify it explicitly within the Astra*Kahn* application code. The pattern can be extracted from the source code of those synchronisers by program analysis.

In order to suppress the growth of the replica chain in the serial replication network, Astra*Kahn* introduces a so-called reverse fixed point mechanism. A reverse fixed point on a channel is a state of a replica, in which it transmits messages from that channel unchanged. Once the Astra*Kahn* runtime system detects that a replica is in the reverse fixed point state on some channel, the connection on that channel can be optimised by removing the replica. Since synchronisers are the only stateful components in Astra*Kahn* , a state of a replica, and thus a reverse fixed point, is formed by the states of its synchronisers. The reverse fixed point can be extracted by program analysis from the source code of the replicas synchronisers.

## 1.2 Contribution

This thesis focuses on the implementation of synchronisers and their role in the serial replication wiring pattern. We provide some minor syntax improvements and detailed explanation how each language construct should be used. An Astra*Kahn* synchroniser has a non-deterministic behaviour. We give an execution algorithm that defines the ordering of non-deterministic choices made by the synchroniser, and which is a basis for the synchroniser runtime. We implement the language compiler that generates the data structure to be interpreted by the runtime and the communication passport of the synchroniser. The compiler performs semantic and type checking and reports source code errors.

The language report [1] explains the machinery behind the serial replication briefly. The serial replication is an important part of Astra*Kahn* and it has to be well-established for the execution environment to be implemented. We analyse the original synchroniser-based approach to fixed points and show how it can be implemented. As a part of the analysis, we give the topological properties of the operand network that are required for the existence of forward and reverse fixed points. The analysis shows that the construction and the debugging of an operand network with a complex fixed point condition can be quite complicated. In order to avoid having to construct complicated operand networks, we provide a simple alternative solution for forward fixed point. Finally, we provide algorithms for the Astra*Kahn* compiler to detect both kinds of fixed point in the operand network.

## 1.3 Outline

The remainder of the thesis is as follows. Chapter 2 introduces Astra*Kahn* and presents some theoretical background in coordination programing and stream processing. Astra*Kahn* is compared with a recent component system example from either field.

In Chapter 4 the implementation of the synchroniser is described in detail. We describe a synchroniser mathematically in order to define its behaviour more precisely. The chapter focuses on the language guide, the execution algorithm and the implementation of the compiler.

In Chapter 5 the machinery behind the serial replication in Astra*Kahn* and the role of synchronisers in it is explained.

Chapter 6 concludes the thesis, providing directions for further research.

# Chapter 2

# Related Work

In this chapter we provide relevant theoretical background in coordination programming and stream processing. We pick a recent component-system example from each field. Then, we describe the combined approach to coordination programming and stream processing, implemented in Astra*Kahn* predecessor S-NET. We sum up the chapter with a comparison of the three components systems, including their approaches to synchronisation.

## 2.1 Coordination Programming

The coordination paradigm offers a promising way to address some issues related to the development of efficient parallel systems. Programming a parallel system can be seen as a combination of two activities: the actual computing part comprising a number of processes that manipulate data and a coordination part that is responsible for communication between the processes.

In the main, coordination is managing dependencies between components. Since the computation is completely separated from the coordination, the processes that comprise the former are seen as black boxes. The programming languages used to write the computational code do not play an important role in setting up the coordination scheme.

Existing coordination models[1] are described in details in the survey [2] by G. Papadopoulos and F. Arbab. They argue that these models fall into two major categories of coordination programming, namely either data-driven or control-driven.

[1] A coordination model encompasses entities being coordinated, a means of coordination and a semantic framework

The main characteristic of data-driven coordination models is that the coordination code is mixed with the process definition. A data-driven coordination language typically offers several coordination primitives which are intertwined with the purely computational code. Many data-driven coordination models have evolved around the notion of shared dataspace. The shared dataspace plays a dual role, being a global data repository and an interprocess communication system. The processes communicate by writing to the shared dataspace and retrieving data from it. Hystorically the first member of this family is LINDA [3]. Strictly speaking, not all data-driven coordination models follow the above pattern of coordination. Some of them use a message-passing based mechanism (MPI, [4]).

Opposite to the data-driven coordination model, the control-driven coordination achieves almost complete separation of concerns between computation and coordination. This is usually achieved by defining a special language that offers facilities for controlling synchronisation, communication, creation and termination of computing components. One of a contemporary members of this family is REO [5]. In REO computational components communicate via complex coordinators, or *connectors*. An undirected channel is an atomic connector in REO. Channels are typed, however, no fixed set of types is assumed. The channel type defines the behaviour of the channel with respect to data. A list of common types is as follows:

- **Sync** A channel of type *Sync* atomically gets data from the input and propagates it to the output
- **Lossy Sync** Same as *Sync*, but data can be lost if the output is not ready to accept data
- **FIFO(n)** A channel of type *FIFO(n)* gets data from the input, temporarily stores it in an internal buffer of size $n$, and propagates it to the output whenever it is ready to accept the data
- **Sync Drain** A channel of type *Sync Drain* atomically gets data from both inputs and loses it
- **Filter(c)** A channel of type *Filter(c)* atomically gets data from the input and propagates it to the output if the filter condition $c$ is satisfied. Otherwise, the data are lost.

Channels are connected with *nodes*. Nodes have fixed merger-replicator behaviour: the data of one of the incoming channels is propagated to all outgoing channels, without storing or altering it. If multiple incoming channels can provide data, the node makes a nondeterministic choice among them. A complex connector in REO is represented as an undirected graph of channels and nodes. C. Baier et al. propose *constraint automata* as an operational model for component connectors in REO [6].

## 2.2 Stream Processing

A stream processing system is a system comprised of a collection of isolated processes that compute in parallel and communicate data solely via static channels. The processes are usually divided into three classes: sources that create data for the system, filters that perform some computation, and sinks that pass data from the system. Stream processing systems are usually visualised as directed graphs.

An overview of stream processing is given in the survey by R. Stephens [7]. Stephens identifies that the earliest type of stream processing systems is dataflow. In the first dataflow programming language LUCID [8], each variable is represented as an infinite stream of values. Computation is carried out by defining transformation functions that process such streams. Lucid is possibly the first language to introduce the idea of filter.

A significant result for concurrency engineering is Kahn's work [9], which outlines the semantics of a simple parallel programming language. Kahn suggests a distributed model of computation where a group of deterministic sequential processes communicate via unbounded FIFO channels under the following assumptions:

- Channels are the only way for processes to communicate
- Channels transmit messages within a finite time
- At any given time a process is either performing computation or waiting for messages on a specific input channel.

Kahn proved that the output of the resulting process network is deterministic, i.e. it does not depend on the ordering of computations at different nodes. The model is commonly referred to as Kahn Process Network (KPN).

A Kahn process may have multiple input and multiple output channels. Reading from a KPN channel is blocking, i.e. a process that reads from an empty channel stalls and can only continue when the channel contains sufficient data. On the contrary, writing to a channel is non-blocking, and it always succeeds since the capacity of a KPN channel is unlimited. Processes cannot test an input channel for data availability without committing to consume the data. KPNs allow arbitrary wiring, i.e. a network may have feedback communication.

In KPNs the number of data elements a process might read from a channel or write to a channel is not restricted. In synchronous dataflow (SDF, [10]) the consumption and production rates of a process are fixed. A recent SDF language is StreamIt [11]. The basic unit of computation in StreamIt is a user-defined single-input single-output (SISO) block called a filter. The filter can communicate with neighbouring blocks via FIFO channels. StreamIt structures an application using the following primitives:

- *Pipeline* specifies sequential composition of filters
- *SplitJoin* specifies parallel composition of filters
- and *FeedbackLoop* provides a way to create loop constructs in a streaming network.

A StreamIt program is a hierarchical composition of these constructs.

Thanks to the single-input and single-output restriction, a filter does not need to synchronise data on multiple input channels and to split result between output channels.

## 2.3 S-Net

S-Net [12], [13] is a declarative coordination language based on stream processing. It defines the behaviour of stateless asynchronous components (*boxes*) that interact with each other in a streaming network. Boxes are written in conventional languages that are subject to contract with S-Net. They execute fully asynchronously, i.e. a box may consume data as soon as it is available from the input stream. Moreover, boxes are SISO, therefore S-Net achieves a near-complete separation of concerns between communication and computation.

Data on streams are organised as variant records of label-value pairs. S-Net provides a special facility, called *synchrocell*, that merges one or more records into a single one. A synchrocell maintains an internal state in order to keep incoming records which match one of the patterns until all patterns have been matched. Then the records are merged into a single one and sent to the output stream.

Streaming networks are expressed in a hierarchical manner using a fixed set of five combinators, viz. serial composition, parallel composition, serial replication, parallel replication and feedback loop. Network combinators are unary or binary operators that take either atomic components, e.g. boxes or synchrocells, or networks as their operands and construct a SISO network; hence, network construction is inductive. Four of the five combinators have non-deterministic versions that permit arbitrary reordering of output streams.

### The type system of S-Net

The type system of S-NET is based on non-recursive variant records with record subtyping. A type is S-NET is a non-empty set of anonymous record variants, and a record is a possibly non-empty set of record entries. Record entries are either *fields* or *tags*. A field consists of a label associated with an opaque value at runtime. A tag is a named integer used for controlling the flow of records through a network. Record subtyping in S-NET is based on the understanding that a subtype is more specific than its supertype. Informally, a type is a subtype of another type if it has additional record entries in the variants or additional variants.

A box or network in S-NET accepts records of a certain type; thus, records upon entry to a certain box or network are up-coerced to its input type. In order to avoid the loss of record entries that are not manipulated by the box, S-NET employs a so-called *flow inheritance* mechanism. Any field or tag of an incoming record that is not explicitly named in the input type of a box or network bypasses the box or network and is added to any outgoing record created in response, unless that record already contains a field or tag with the same label.

### Components

- **Boxes**

  Boxes are the atomic building blocks of streaming networks in S-NET. User-defined boxes can be specified in any conventional programming language that has an interface with S-NET. Generally, a user-defined box may produce a variable number of output records in response to a single input record, which is up to the box implementation. S-NET requires to specify the box type signature, which describes the typewise stream-to-stream transformation performed by the box.

  S-NET provides so-called built-in filter boxes (or *filters*), which allow various house-keeping operations that do not require knowledge of field values. Those operations include, but not limited to, elimination of fields and tags from records, copying fields and tags, adding tags, and splitting records.

- **Synchrocells**

  S-NET provides a built-in component called *synchrocell* for the purpose of joining of multiple records appearing in some order on a single input stream. Syntactically, a synchrocell consists of one or more patterns that match incoming records. A match happens when the type of a record is a subtype of the type pattern. The synchrocell provides storage for exactly one record of each pattern, and records with an already matched pattern are forwarded directly to the output stream. Once all the patterns have been matched and a merged record has been sent on to the output stream, the synchrocell serves as an identity box, i.e. it forwards all incoming records to the output stream.

The paper [14] shows that in conjunction with other S-Net features, the simplistic synchrocell design allows the implementation of essential synchronisation features making the synchrocell an efficient synchronisation facility for asynchronous data flow computing in the style of S-Net. For example, continuous synchronisation, which is a common kind of synchronisation in streaming networks, can be implemented by applying a serial replication combinator to a synchroncell. Also, the paper ([14]) demonstrates modelling of a stateful computation using the property of synchrocell to become an identity once it has performed joining.

### Network Composition

Network composition in S-Net is an inductive process with boxes as base cases. S-Net networks are constructed hierarchically using a set of five network combinators. Network combinators are either unary or binary, and they create compound networks that have a single input and a single output stream. Routing decisions are made at split points of a network and are based upon the type of the subnetworks and the type of the actual record.

**Serial composition** applies to two operands. It connects the output stream of the first operand to the input stream of the second operand. The input stream of the first operand and the output stream of the secord operand become those of the combined network.

**Parallel composition** applies to two operands and combines them in parallel. An incoming record is sent to exactly one operand depending on its own type and the type of records accepted by either of the operands. The output streams of the operands are merged into a single stream, which becomes the output stream of the combined network.

**Serial replication** applies to two operands. It creates infinitely many copies of its first operand, which is a network, and connects those copies by serial composition. The second operand is a type pattern, such that each record that is a subtype of this type pattern leaves the replication pipeline through the output stream.

**Indexed parallel replication** applies to two operands. Similar to serial replication, it creates infinitely many replicas of the first operand, which is a network, and connects the replicas by parallel composition. The second operand is a tag. Each incoming record must have this tag and is sent to the replica with the tag value in the record.

**Feedback loop** applies to two operands. The first operand is a network and the second operand is a type pattern. Records which are input to the feedback loop network enter the operand network unconditionally. All output records of the operand network that are a subtype of the type pattern are fed back to the input of the feedback loop network. All other output records leave the feedback loop network.

Serial and parallel replication network are demand-driven, hence the replicas are created dynamically on runtime.

All combinators except for the serial composition have non-deterministic and deterministic variants. Deterministic variants of combinators preserve the ordering of records in the output stream, while non-deterministic variants are allowed to completely reorder it.

## 2.4 Summary

Earlier on we reviewed recent component systems based on coordination programming (Reo) and stream processing (StreamIt), and described the approach to component coordination developed in S-Net. The stream processing based approaches StreamIt and S-Net impose structuring on networks with fixed sets of combinators, while the coordination language Reo only supports unstructured component connection. In Reo, the computational components are connected into a network with complex connectors that are constructed of channels typed with respect to their synchronisation properties. Just like the Reo's approach to data synchronisation, S-Net achieves a near-complete separation of concerns between computation and communication. However, in S-Net, a computational component's interface is restricted to SISO. Additionally, S-Net provides a stream synchronisation facility called synchrocell. Similar to S-Net, the computational components in StreamIt are SISO. StreamIt is based on the synchronous dataflow model, where neighbouring components communicate synchronously.

In order to support dynamic reconfiguration of streaming networks, S-Net requires computational components to have no state. A heuristic scheduler that utilises positive and negative demands of the stream communication was developed for S-Net in [15]. The ability to dynamically reconfigure a streaming network opens opportunities for parallelisation. StreamIt does not require the components to be stateless; it relies on the static scheduling of the synchronous data flow programs. Reo is clueless about the components it coordinates; it focuses on the components connection, rather then on the components themselves. In other words, the problem of automatic parallelisation is not set for both StreamIt and Reo.

# Chapter 3

# Astra*Kahn*

In this chapter we present the concepts of a new coordination language Astra*Kahn* . Astra*Kahn* is an attempt to provide a component system with automatic concurrency management. Astra*Kahn* defines the coordination behaviour of fully asynchronous components (boxes) and their orderly interconnection via stream-carrying bounded FIFO channels. In Astra*Kahn* data on streams are organised as sequences of messages. Each message conforms to one or more statically known formats.

Astra*Kahn* provides a facility for stream synchronisation in the form of a special component called a *synchroniser*. The behaviour of the synchroniser is not fixed; instead, it is defined in a dedicated language that is a part of Astra*Kahn* paradigm. Similar to S-NET, boxes are stateless, hence they do not synchronise data; this work is done by synchronisers.

Astra*Kahn* structures streaming networks using a total of four combinators, namely: the serial connection, the parallel connection, the wrap-around connection and the serial replication. Network combinators may take either boxes or networks as their operands, hence the network construction is an inductive process.

In the following sections the concepts of Astra*Kahn* are explained in more detail.

## 3.1 Channels and Messages

### Channels

Channels in Astra*Kahn* are named FIFO queues with a limited capacity. A channel carries a segmented stream that consists of message sequences and those may in turn consist of sequences in their own right. In order to mark the beginning and end of a sequence, Astra*Kahn* supports a special kind of message called a segmentation mark.

Segmentation marks can be thought of as brackets. Astra*Kahn* requires that a stream of message sequences that flows through a channel has a static bracketing depth. Therefore, each message on a given channel is found between the same number of brackets. The sequence of messages starts with a certain number of opening brackets and ends with the same number of closing brackets. Within the sequence brackets can occur only in the following combination:

$$\underbrace{)\dots)}_{k}\underbrace{(\dots(}_{k},$$

where $k \leq d$, and $d$ is the number of opening brackets in the beginning of the stream. This combination constitutes the segmentation mark $\sigma_k$. The bracketing depth $d \geq 0$ is a static characteristic of a channel[1].

## The Type System of Astra*Kahn*

The type system of Astra*Kahn* is based on the Message Definition Language (MDL, [1]) which is a language of abstract terms that are built recursively from the ground up. Structurally the terms are symbolic trees with the following kinds of leaf:

**Symbol, Number, String** terms represent a certain finite quality

**Variable** term ranges over terms

**Flag** is a boolean variable that only occurs in a certain context.

Other terms are built recursively using the following types of constructor:

**Tuple** is a sequence of terms in linear order

**List** is an extensible sequence of terms in linear order

**Record** is an extensible collection of label-term pairs

**Choice** is an extensible collection of alternative labeled terms

**Switch** is a collection of guarded terms that represents exactly one of them depending on the value of the boolean guards

[1] Indeed, the bracketing depth of a channel that would carry the stream of message lists

$$(((\,a\,\underbrace{)(}_{\sigma_1}\,b\,\underbrace{))((}_{\sigma_2}\,c\,\underbrace{)(}_{\sigma_1}\,d\,)))$$

is 3

Data on streams are organised as sequences of messages defined by a **choice** of **records**, which is similar to S-NET. However, in Astra*Kahn* the data carried by a record entry can be of any format that MDL allows. Also, in Astra*Kahn* a choice that is known to carry a single record is labeled *uniq*. Unlike S-NET, Astra*Kahn* does not need to decide on message routing since the channels are named, hence Astra*Kahn* messages do not need tags.

An Astra*Kahn* component, either a box or a synchroniser, is both a consumer and a producer for some other components in the network. Hence to guarantee the static correctness of a connection, the subtyping relation between the consumer's input and the producer's output types must be satisfied. In order to check the static correctness over the network, a component can be abstracted with respect to its data-transformation behaviour as an implicative statement $p \Rightarrow P$, called a passport, where $p$ is the type of the input message and $P$ is the type of the output message. During the check, the Astra*Kahn* compiler extracts the topology of the network, forms the subtyping relations between the passports and performs constraint solving in order to instantiate all term variables. If the constraint system is satisfiable, then the whole program is consistent and type correct.

## 3.2 Components

### Boxes

Boxes are the atomic building blocks of Astra*Kahn* networks that perform the computation. An Astra*Kahn* box is deterministic in the sense that for every partial input stream it produces a deterministic output stream[2].

Conceptually, boxes can be specified in any conventional programming language; however, they are subject to a contract that defines acceptable behaviour for boxes. Any guarantees that Astra*Kahn* offers are subject to the fulfilment of the contract on behalf of all the boxes. The interface between a box and the Astra*Kahn* runtime system is defined by the Astra*Kahn* Box-API for each supported box language.

Unlike S-NET, which does not require to specify any box properties but its type signature, Astra*Kahn* declares seven box categories with respect to their algebraic properties and effect of channel segmentation[3]:

[2] For a function $f(x) : \mathcal{I} \to \mathcal{O}$, where $\mathcal{I}$ is the totality of $f(x)$ input streams and $\mathcal{O}$ is the totality of $f(x)$ output streams, $\forall p \in \mathcal{I} \wedge \forall t : p \cup t \in \mathcal{I} \ : \ f(p \,||\, t) = f(p) \,||\, f(t)$

[3] The box code does not see the segmentation marks; Astra*Kahn* deals with them all by itself

**Transductor** A transductor has one input channel and one or more output channels and responds with no more than one output message on each of its output channels. Segmentation marks are passed on to all the output channels of the box, bypassing the box code.

**Inductor** An inductor has one input channel and one or more output channels and responds to a single message from the input channel with a sequence of messages on each of its output channels. Before the input stream is passed to the inductor, each $\sigma_k$ in it with $k > 0$ is replaced by $\sigma_{k+1}$, and a $\sigma_1$ is inserted between every two consecutive data messages. Segmentation marks are bypassed from the input to all the output channels by the coordinator when encountered at the input of the inductor.

**Reductor** A reductor implements the reduction operation for a list of input messages. The reductors can have more than one output channel with one of them reserved for the results of the reduction. Astra*Kahn* classifies reductors by the number of input channels and properties of the reduction operation they implement. There exist five classes of reductor:

- **Dyadic ordered** A dyadic ordered reductor has two input channels. The first input channel is reserved for the initial value. The reduction operator is applied to the messages in the order that they arrive on the second input channel
- **Dyadic unordered** Same as dyadic ordered except that the reduction operator can be applied to the messages on the second channel in any order without affecting the result
- **Monadic ordered and monadic unordered** Same as dyadic reductors except monadic reductors have one input channel. A monadic reductor is only started when two messages are received
- **Monadic segmented** A monadic reductor recursively processes an input list of messages that can be segmented into arbitrary sublists until the list is reduced to a single message

An AstraKahn box generally is not SISO, which makes a difference from S-Net. Typically it has a single input channel, however, the number of output channels, although statically known, is not restricted.

### Synchronisers

Synchronisers are non-deterministic finite state machines for joining messages and sending them on to the output channels. Astra*Kahn* provides synchronisers with memory for storing received messages.

A synchroniser can have any number of input and output channels. Unlike boxes, synchronisers maintain an internal state and generally accept messages from an input channel in certain states, while in another state the channel may be blocked until a state transition brings the synchroniser to a state in which messages from the channel are accepted.

A synchroniser can also compute trivial extensions for messages. For example, it can append a labeled integer value to a message. It also detects segmentation marks in an input stream and can change the segmentation of the stream by sending segmentation marks to the output channels.

The state transitions of a synchroniser can depend on the content of the current message but never on that of a stored one. In order for the synchroniser to read values from the current message, it is matched with a pattern specified within the triggering condition of the transition. In addition, the triggering condition may check the matched values if they are known to be integers. If the message was matched and the integer values satisfy the condition, then the transition fires.

The act of sending a message to an output channel is associated with a transition. Once the transition is known to fire, the synchroniser computes the message extensions, combines all the parts of the message together and sends it on to the output channel.

Astra*Kahn* provides a dedicated language of synchronisers.

Astra*Kahn* does not introduce the SISO restriction for boxes; instead, it provides a more complex stream synchronisation facility. Unlike the Reo connector and the S-Net synchrocell, the Astra*Kahn* synchroniser is able to process messages, e.g. read and change their content to some extent. In S-Net this can be implemented with filters.

## 3.3 Network Composition

The construction of streaming networks in Astra*Kahn* is hierarchical: components are wired into a subnetwork, which in turn can act as a component in a larger network, etc. In order to wire the components, Astra*Kahn* provides a set of wiring patterns sufficient to achieve arbitrary wiring [1]. Each pattern identifies input/output channels of the operand(s) with one another and with the input/output channels of the result.

Three patterns are static, applicable to one or two operands:

**Serial connection** applies to two operands. All outputs of the first operand are wired to identically named inputs of the second operand if they exist. The rest of the inputs and outputs contribute to the input/output sets of the resulting network.

**Parallel connection** applies to two operands. Two operand networks are placed side by side without connection and their input and output channels form the input and output channel sets of the resulting network.

**Wrap-around connection** applies to a single operand. Each output channel of the operand that matches an input channel by name is wired to it with a special wrap-around channel, thus completing a cyclic connection. In order to avoid deadlocks, Astra*Kahn* does not limit the capacity of wrap-around channels; their capacity is only limited by the amount of memory available for the queues in the system[4].

These three patterns are sufficient to achieve arbitrary wiring of the components.

The fourth pattern – **serial replication** – replicates the single operand network infinitely and wires up the replicas with the serial connection. In implementation, actual replication is demand-driven: the chain of replicas is extended dynamically. Messages are extracted from the chain and sent to the output channel when they satisfy the fixed point condition, see Chapter 5.

Astra*Kahn* attempts to provide a component system with automatic concurrency management based on communication demand. Communication demand is observed when a box produces messages to its output channel slower than they can be consumed from that channel. If the input channel of the slow box is not empty, several copies of the box may run in parallel to produce output messages faster and to minimize the latency of the application network. Otherwise, the communication demand is propagated backwards to a box that produces messages for the slow box. Automatic box replications in Astra*Kahn* are demand-driven; however, it is up to the Astra*Kahn* runtime how many copies of each box to run at a time.

Astra*Kahn* does not have the parallel replication combinator which exists in S-Net; the parallel replication is implemented by the concurrency management mechanism. The synchronisation in the sequences of messages produced by the parallel replicas can be implemented by an array of synchronisers that are indexed within the declared limits.

[4]if there is no memory available, the application crashes

# Chapter 4

# **Astra***Kahn* Synchroniser

In this chapter we describe the source language for an Astra*Kahn* synchroniser and its implementation. Prior to describing the language in details, we present a mathematical model of a synchroniser from [1] and outline sources of non-determinism in synchronisers.

The *aksync* compiler is integrated into the current Astra*Kahn* runtime system prototype. It takes the source code of a synchroniser program, performs the syntactic and semantic analysis of it, builds a data structure for the Astra*Kahn* runtime and generates the communication passport of the synchroniser.

## 4.1 Mathematical Model

Synchronisers are finite state machines for joining received messages and sending them on to the output channels. Synchronisers can have one or more input and output channels. From the mathematical point of view a synchroniser is a pair

$$(\Phi,\ \Pi),$$

where $\Phi = (A,\ S,\ T)$ is a nondeterministic state machine with the alphabet of events $A \subseteq C \times P$, where $C$ denotes the set of input channels and $P$ the set of the predicates on channel messages. An event $(c,\ p) \in A$ represents the reception of message $\mu$ on channel $c$ that satisfies the predicate $p$. The set of abstract states in $\Phi$ is denoted as $S = \{s_0, \ldots, s_k, \ldots, s_n\}$, $k \leq n$, with the start state $s_0$, and the transition matrix as $T\ :\ A \times S \to S$.

The path functional $\Pi(S,\ \Omega)$, where $\Omega$ is the set of output channels, defines the output of the synchroniser. The function $\Pi\,(s_k,\ \omega_m) \in \Pi$ defines the output to channel $\omega_m \in \Omega$ of the synchroniser in state $s_k \in S$. In state $s_k$ the functional is based on the retrospective sequence of transitions from the most recent visit to the start state $s_0$ to $s_k$:

$$(s_0, a_0),\ (s_1, a_1), \ldots (s_k, a_k),$$

where $a_i = c_i \times p_i \in A$, $0 \leq i \leq k$ is the alphabet symbol that caused the transition from the state $s_i$. Let $\mu_i$ be the message received in the transition from the state $s_i$. Then

$$\Pi\,(s_k, \omega_m) = \psi_\sqcap\ \{\mu_i \mid \rho^m_{ki}\ (s_i),\ 0 \leq i \leq k\}, \tag{4.1}$$

where $\rho^m_{ki}$ is the selection predicate that defines $\Pi$, and $\psi_\sqcap$ is the operator that coerces the messages in the operand set to their joint greatest subtype. A synchroniser in Fig. 4.1 emits to channel $\omega_m$ only a message received on channel $c_k$ in state $s_k$. Fig. 4.1 shows the path functionals for $\omega_m$ that correspond to each transition on channel $c_i$ from state $s_i$, $0 \leq i \leq k$.

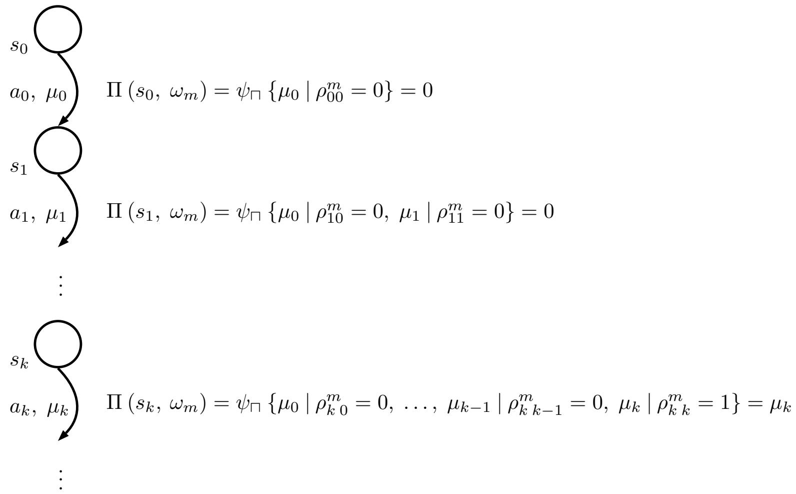


FIGURE 4.1: The path functionals for output channel $\omega_m$

From the above, the synchroniser is fully defined by two functions:

1. The transition matrix $T$

   The state machine can have a regular structure whereby many transitions can be defined at once by a formula with some limited-range integer variables. For example, a machine with 8 states could have a transition matrix defined thus: $S_{k\ mod\ 8} \rightarrow S_{k+1\ mod\ 8}$. In order to be able to employ regular transition graphs, AstraKahn allows synchronisers to declare *state* variables.

   **Example: the counter synchroniser** Counter sends every $n$-th message from its input channel to the output channel, other messages are disregarded. The transition diagram of the counter synchroniser with input channel $a$ and output channel $c$ for $n = 3$ is given in Figure 4.2.

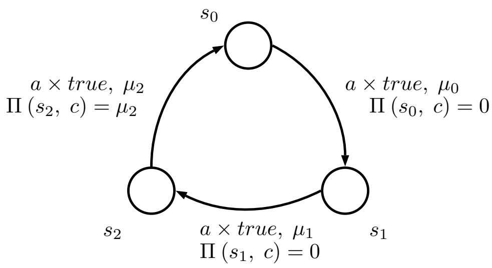


FIGURE 4.2: The transition diagram of the counter synchroniser ($n = 3$)

The transition matrix $T$ of the synchroniser in Fig. 4.2 is

| $A \setminus S$ | $s_0$ | $s_1$ | $s_2$ |
|---|---|---|---|
| $(a,\ true)$ | $s_1$ | $s_2$ | $s_0$ |

The state machine behind the counter has a regular structure, and for this synchroniser all its transitions can be defined with a single formula: $S_{k\ mod\ 3} \rightarrow S_{k+1\ mod\ 3}$. Considering this, the transition matrix $T$ would be:

| $A \setminus S$ | $S_{k\ mod\ 3}$ |
|---|---|
| $(a,\ true)$ | $S_{k+1\ mod\ 3}$ |

. The diagram 4.2 represents the unrolled regular structure of the synchroniser. However, this representation is inconvenient when $n \gg 1$. The diagram can be folded using state variables. Two possible variants are shown in Fig. 4.3. The state variable $i$ acts as an induction variable in a while loop with the exit condition $c \geq 3$.

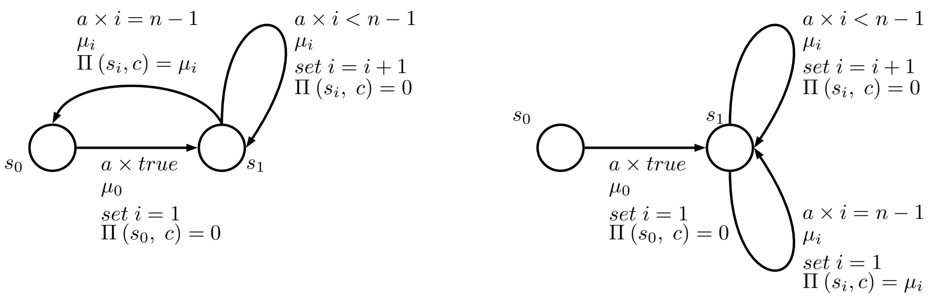


FIGURE 4.3: The transition diagrams of the counter synchroniser

2. The selection predicate $\rho$ (see formula 4.1)

   In a given state $s_k$ for each output channel $\omega_m$ we note all $\rho^m_{ki}$ that are true. The corresponding message values must be saved in previous states and recalled in state $s_k$. It is expected that the boolean vector $[\rho^m_{ki}]$ has only very few true elements. Although the functional $\Pi$ can be implemented as function that retrieves all the saved messages at once, it is feasible to implement it in the form of individual *store* variables. A store variable is associated with an input channel.

   **Example: the binary zip synchroniser** Zip2 receives messages on its input channels and sends their concatenation to the output channel. In the resulting concatenation there is exactly one message from each input channel and those messages are combined for the output. The transition diagram of the zip2 synchroniser with input channels $a$ and $b$ and output channel $c$ is given in Fig. 4.4. Store variables $ma$ and $mb$ are associated with the input channels $a$ and $b$ respectively.

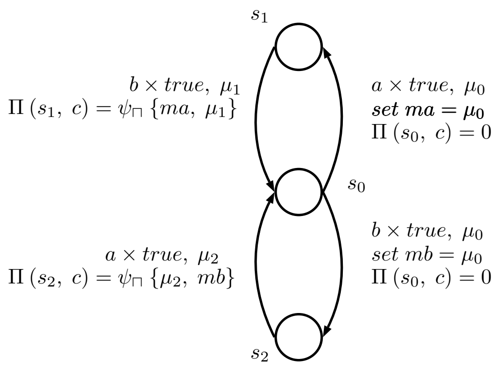


FIGURE 4.4: The transition diagram of the zip2 synchroniser

So far we have given a formal definition of the synchroniser. Along with it we have introduced two important concepts for the synchroniser that are considered as a part of the synchroniser language. First is a message storage mechanism we call a store variable. Second is a mechanism for defining large regular transition matrices, which we call a state variable. In the next section we describe the programming language of synchronisers. We shall note that the language allows some operations on store variables[1] that the presented mathematical model cannot describe. The model should be fixed in accordance with the synchroniser language in the future.

## 4.2 The Language of **Astra***Kahn* Synchronisers

The Astra*Kahn* synchroniser is a finite state machine, therefore the basic building blocks of a synchroniser program are states and transitions. A state of a synchroniser is fully defined by the corresponding state of the finite state machine and the values of the state variables. A transition is the act of moving to another state which is initiated by a triggering event. A triggering event for the synchroniser transition is the arrival of a message to the associated channel. The message may be required to have a specific structure. In addition, a transition may be guarded by special conditions on state variable values. If the condition is satisfied the transition fires, otherwise it is cancelled.

[1]E.g. merging two store variables into another one, appending a message extension that was computer by the synchroniser into a store variable, etc.

Once a transition is known to fire, optional actions may be performed before the underlying state machine makes the move. These actions include changing the state and store variables and sending messages to output channels. In order to change the state and store variables, the synchroniser language provides state and store expressions over them.

This section gives an overview of the Astra*Kahn* synchroniser programming language. The formal grammar of the Astra*Kahn* synchroniser is provided in Appendix A.1.

### Program structure

A synchroniser program consists of a header followed by a body wrapped in braces. The beginning of a synchroniser program is indicated by the keyword *synch*.

The header contains the synchroniser's name and the channel signature. The name is an ASCII string that follows the C convention.

The body lists the state and store variable declarations and the states of the underlying finite state machine. Each state is supplemented by a list of transitions. Each transition declares its triggering condition which includes an optional guarding state expression, an optional list of actions, and, finally, the destination state.

### Channel signature

The channel signature defines the input and output channels of the synchroniser and their bracketing depths. The synchroniser header (Fig. 4.5) declares the synchroniser *min* with two input channels that are connected to the ports $a$, $b$ and two output channels that are connected to the ports $c$, $d$. If the bracketing depths of the channels are not specified, they are assumed to be 0. Thus, the bracketing depth of the channel $a$ is 0.

The input channel depth $-1$ indicates that the input channel is ignored in the synchroniser program. The output channel with the depth $-1$ must not have data sent to them.

```
synch min (a, b:p | c:2, d:p+1)
```

FIGURE 4.5: The synchroniser header

The Astra*Kahn* synchroniser allows one to declare constant and configurable integer depths for the input and output channels. In addition, the depth of an output channel can be specified with an integer shift to the configurable input channel depth.

The input channels are required to have the bracketing depths specified in the signature. Thus, the channel $a$ (i.e. the channel connected to the port $a$) of $min$ must have zero bracketing depth. The channel $b$ has a configurable bracketing depth $p$. The actual values of configurable bracketing depths of input channels are determined by the Astra*Kahn* compiler by tracing bracketing depth relations over the application network.

The output channels of a synchroniser are guaranteed to have the bracketing depths specified in its channel signature. Thus, the synchroniser $min$ must send messages to the output channel $c$ at depth 2. The output channel $d$ must have the bracketing depth $p + 1$, i.e. the depth of the input channel $b$ plus 1.

### Variable declaration

The start of a state variables declaration is marked by the keyword *state*. A state variable may be either an unsigned integer of some size or a C-style enumeration. State and store variable names are user-defined identifiers. A user-defined identifier is an ASCII string that follows the C convention.

Line 1 in Fig. 4.6 declares state variables $a$, $b$, $c$ of size 4. Thus, all three variables are declared to have integer values in the range $[0; 15]$. Generally, a state variable of size $n$ has integer values in the range $[0; 2^n - 1]$. State variable $c$ is initialised with an integer value 0.

The state variable $foo$ that is declared on the line 2 in Fig. 4.6 can only be assigned the values $d$, $e$ and $f$ specified in the enumeration. The enumeration values are integer constants. If they are not specified explicitly, they are assigned consecutive positive integers starting with 0. Thus, the variable $foo$ has integer values $d = 0$, $e = 1$ and $f = 2$.

The values can be specified explicitly (see line 3 in Fig. 4.6)

Integer state variables and enum values can be mixed freely in state expressions. Enum values are interpreted as integer constants.

```
1 state int(4) a, b, c = 0;
2 state enum(d, e, f) foo;
3 state enum(x = 1, y = 2, z = 4) bar;
4 store msg_a, msg_b;
```

FIGURE 4.6: State and store variables declaration

A store variable declaration begins with the keyword *store*. Line 4 in Fig. 4.6 declares state variables $msg\_a$ and $msg\_b$. Store variables do not need an explicit type specification; their types are determined on the first assignment to the variable.

All the state and store variables are global to all the states.

### States and transitions

States and transitions of the synchroniser define which channels are read and in what order. Fig. 4.7 presents the code of the binary zip synchroniser's state machine. Line 1 declares the start state of the synchroniser. The *on* clause indicates the beginning of the transition list. In the start state the zip2 synchroniser accepts messages from both input channels $a$ and $b$. State and store expressions associated with the transition and the destination state are specified in the braces.

```
1  start {
2    on:
3      a { goto s1;     }
4      b { goto s2;     }
5  }
6  s1 {
7    on:
8      b { goto start; }
9  }
10 s2 {
11   on:
12     a { goto start; }
13 }
```

FIGURE 4.7: State machine of the zip2 synchroniser

When the zip2 synchroniser is in the start state and it receives a message from channel $a$, the underlying state machine transitions to state $s1$. In this state the synchroniser can only receive messages from channel $b$ since there is no transition triggered by channel $a$ and defined in this state. When the message on channel $b$ is received, the state machine transitions to the start state. Lines 10-13 define similar behaviour in state $s2$.

The synchroniser language supports top-down prioritised transition scopes. They are marked by the $elseon$ keyword. A synchroniser in state $foo$ in Fig. 4.8 accepts messages from channels connected to the ports $a$, $b$, $c$ and $d$. When no destination state is specified for a transition, a synchroniser transitions to the current state. If all channels are ready at the same time in state $foo$, the synchroniser processes messages from either channel $a$ or $b$ first. When all messages from channels $a$ and $b$ are processed the synchroniser receives messages from channel $c$. If there are no messages in channels $a$, $b$ and $c$ the synchroniser receives messages from channel $d$.

### State expressions

A state expression is a combination of integer constants, state variables and operators, which defines an integer value. The interpretation of a state expression follows the C

```
foo {
  on:
    a { }
    b { }
  elseon:
    c { }
  elseon:
    d { }
}
```

FIGURE 4.8: Prioritised transition scopes

rules of precedence and association. State expressions can be assigned to state variables. Under the assumption that the output channel has the infinite capacity a synthetic example in Fig. 4.9 counts the number of messages received from channel $a$ between the arrivals of messages in channel $b$. Line 1 declares the 8-bit integer *count* and initialises it with 0. When a message from $a$ is received the value of *count* increases by 1 (see line 5).

```
1  state int(8) count = 0;
2  foo {
3    on:
4      a {
5        set count = [count + 1];
6      }
7    elseon:
8      b {
9        set n = [count], count = [0];
10       send count:[n] => c;
11     }
12 }
```

FIGURE 4.9: Use of state variables and expressions

When a message from channel $b$ is received the value of *count* is stored in the temporary variable $n$, set to 0 and then $n$ is sent to the output channel.

The variable $n$ does not have to be declared and is considered an alias for the integer expression. Temporary variables are available until the state machine of a synchroniser makes the next transition.

### Triggering of a transition

The channel name on its own stands for the availability predicate for the corresponding channel, i.e. the condition that a message of any kind is available. Whether a transition takes place depends on the channel status and optionally the content of the message.

When a message is received on a channel, it can be matched with a pattern in order to extract parameters needed to select a specific transition. Line 3 of Fig. 4.10 checks whether a message received from the input channel $a$ is a unique choice. Then the one and only variant is checked on whether it contains the label $x$. If it does, the value of $x$ is stored in a temporary variable $x$. The tail of the message, i.e. all label-value pairs except for the value labeled $x$, is stored in a temporary variable $t$. Both $x$ and $t$ are available until the state machine makes the next transition.

```
1 foo {
2   on:
3     a.(x || t)   { }
4     a.?v         { }
5     a.?v(x, y)   { }
6     a.@[k]       { }
7 }
```

FIGURE 4.10: Message content extraction

To support message formats where several variants of a message are possible, a qualifier **?**$\alpha$ is available as an input condition. It qualifies input messages as belonging to the $\alpha$ variant. Line 4 of Fig. 4.10 checks whether the message received from channel $a$ contains the variant $v$. Line 5 checks whether the message that contains the variant $v$ with only two fields labeled $x$ and $y$.

A channel carries a stream that consists of messages and possibly segmentation marks. Line 6 in Fig. 4.10 checks if the message is a segmentation mark of the depth equal to $k$. The depth of a segmentation mark can be compared with a state expression.

Several different channels can be tested in any given state, however, once the readiness of a channel is established, the synchroniser is committed. Hence the set of conditions applied to the message on any input channel must be exhaustive. In Fig. 4.10 it is not, because there is no pattern for messages that do not contain the field label $x$, the variant $v$ and are not a segmentation mark of depth $k$ at the same time. In this case the final clause $a$**.else;** is assumed. This clause discards the input message and transitions the synchroniser back to its current state.

A transition can be guarded by a state expression. In this case the transition fires only if the guarding expression evaluates to true. The synchroniser in Fig. 4.11 sends every 256-th message to the output channel. Line 1 declares the 8-bit state variable $i$ and initialises it with 0. The variable is incremented every time a message from channel $a$ is received, except when it reaches 255, in which case it is reset to 0 and the received message is sent down channel $c$.

Values that are matched from the message can be used in guarding state expressions.

```
state int(8) i;
start {
  on:
    a & [i < 255] {
      set i = [i + 1];
    }
    a & [i = 255] {
      set i = [0];
      send this => c;
    }
}
```

FIGURE 4.11: Use of guarding state expressions

### Store expressions and sending messages

Store expression is a mechanism to combine data. In Astra*Kahn* data are typed. Types are MDL terms. The result of the store expression can be either saved in a store variable or sent down an output channel.

The example in Fig. 4.12 demonstrates the use of store expressions and the *send* clause. In the start state the synchroniser receives messages from channel $a$ that has the label $n$ in it. Line 5 increments the value under the label $n$ and stores it in the store variable $ma$ under the label $n$ together with the tail $t$. The operator $'$ applied to the variable $x$ creates the record $'x' : value(x)$. This is a shorthand useful for the avoidance of tedious notation.

```
1  store ma;
2  start {
3    on:
4      a.(n || t) {
5        set ma = (n:[n+1] || t);
6        goto s1;
7      }
8  }
9  s1 {
10   on:
11     b {
12       send ma || this => c;
13       goto start;
14     }
15 }
```

FIGURE 4.12: Use of store expressions and the *send* clause

A message received on a channel is referred to by the keyword *this* within the active transition. In state $s1$ the synchroniser receives messages from channel $b$. When a message is received, it is concatenated with the store variable $ma$ (see line 12) and sent to the output channel $c$.

## 4.3 Execution Order of Synchroniser

In order to achieve the lowest latency, an Astra*Kahn* synchroniser exploits a non-deterministic behaviour. In a certain state more than one input channel may be ready, however, a state machine receives one input message at a time. The synchroniser does not take any transition that potentially causes sending to a blocked channel. Of the transitions that do not send to blocked channels, which one will be triggered is defined by the fairness policy: when more than one transition is possible in a given state, all choices will be made with the same frequency. If transitions are prioritised, the choices are made within each scope at first. The fair choice implementation is based on the number of transition executions on the runtime. The synchroniser always takes the transition that was taken the least among the available transitions.

Once the transition has been taken and the associated actions have been executed, the synchroniser transitions to the destination state. In order to avoid transitioning to a state in which there are no ready input channels the synchroniser may be provided with a set of destinations to choose. In the synchroniser language this is expressed as a goto-clause with multiple destinations. The algorithm in Fig. 4.13 defines the ordering of the choices that a synchroniser makes during the execution.

## 4.4 The Implementation of the *aksync* Compiler

In this section we describe the implementation of the Astra*Kahn* synchroniser compiler *aksync*. At the current stage of the Astra*Kahn* software stack development the *aksync* compiler is highly integrated into the Astra*Kahn* runtime system prototype. Similar to the runtime system the *aksync* compiler is implemented in Python. It generates an intermediate representation of the synchroniser program which is passed to the synchroniser runtime.

The lexical and syntax analysers are implemented using PLY [16] - an implementation of lex and yacc for Python. The semantic analyser performs semantic and type checking. The code generator emits the synchroniser's runtime data structure and derives its passport.

```
Require: The current state of the synchroniser (curr_state)
Ensure: The state, to which the synchroniser transits from the current state
function RUN(curr_state)
    ReadyInputs ← ready channels that are read in curr_state
    for each channel in ReadyInputs do
        trans ← transitions from curr_state that read from channel
        if ∃ t ∈ trans ∧ t causes sending to a blocked channel then
            remove channel from ReadyInputs
        end if
    end for
    if ReadyInputs = ∅ then
        return curr_state
    end if

    channel ← the least frequently taken channel from ReadyInputs
    message ← fetch a message from channel
    ▷ iterate over transitions sorted by the number of executions from low to high
    for each number of transition executions N in curr_state do
        trans ← all transitions with the number of executions N that read from channel
        valid_trans ← all transitions from trans with satisfied conditions
        else_trans ← the .else transition from trans with the satisfied condition
        if valid_trans = ∅ then
            if else_trans = nil then
                continue
            else
                choose(else_trans)
            end if
        else
            choose(the least frequently taken transition from valid_trans)
        end if
    end for
    if no transition has been chosen then
        return curr_state
    end if
    act()    ▷ perform the actions associated with the chosen transition
    ImmediateStates ← states listed by the goto clause

    for each state in ImmediateStates do
        ReadyInputs ← ready channels that are read in state
        trans ← transitions from state that read from ReadyInputs
        if ReadyInputs = ∅ ∨ ∃ t ∈ trans ∧ t causes sending to a blocked channel then
            remove state from ImmediateStates
        end if
    end for
    if ImmediateStates ≠ ∅ then
        return the least frequently taken state from ImmediateStates
    end if
    return the least frequently taken state from the goto clause list
end function
```

FIGURE 4.13: The execution of a synchroniser

### 4.4.1 Lexical Analysis

#### Lexical analyser

The lexical analyser reads the stream of characters making up the source program and groups the characters into lexemes. For each lexeme, the lexical analyser produces a token of the form $\langle name, value \rangle$, which it passes to the syntax analyser. For tokens that do not need the *value*, such as punctuation, reserved words and keywords, the second component is omitted. Those are given in Appendix A.2.

PLY implements the way in which traditional tools work. Specifically, the Python lex provides an external interface in the form of a `token()` function that returns the next valid token on the input stream. Token positional information, which is useful in the context of error handling, is managed by the Python yacc.

#### Preprocessor

The original synchroniser language in [1] provides integer configuration parameters to avoid having to trivially alter synchroniser programs. They are specified in brackets between the synchroniser's name and its channel signature. A program that has to be compiled with uninstantiated input parameters potentially makes the analyses in the compiler more conservative. Thus, all parameters should be instantiated before compilation starts.

For the sake of simplicity, we implement substitution for the free parameters in a tiny lexical preprocessor. The lexical preprocessor requires only lexical analysis. It substitutes tokenized character sequences for other tokenized character sequences according to some user-defined rules. The preprocessor that has been implemented does not support any directives and only performs macro substitution. The compiler reads macros from its invocation command and then passes them to the preprocessor.

With the above implementation, configuration parameters do not have to be specified in the synchroniser program. However, such an implementation of configuration parameters has a serious drawback: a macro defines a blind substitution. There is no way to make sure that it defines a rule for the substitution of an integer parameter because lexical analysis does not know anything about program structure and semantics. However, at least rules that are obviously not suitable for integer parameter substitution can be diagnosed.

### 4.4.2 Syntax Analysis

#### Syntax analyser

The syntax analyser obtains a string of tokens from the lexical analyser and verifies that it can be generated by the grammar of the source language. The syntax analyser is expected to report syntax errors in an intelligible way and recover from common errors in order to continue processing of the remainder of the program. For well-formed programs, the syntax analyser constructs an intermediate representation of the program and passes it to the rest of the compiler for further processing. The *aksync* implementation is based on the abstract syntax tree representation.

The Python yacc generates a syntax-directed translator. Syntax-directed translation is done by attaching program fragments to productions in a grammar. The program fragment or so-called semantic action is executed when the production is used during syntax analysis. The combined result of these executions produces the intermediate representation of a program in the order induced by the syntax analysis. The Python yacc accepts source language syntax specification in context-free grammar form. The grammar for the synchroniser language is given in Appendix A.1.

The syntax error handling and recovery mechanism of PLY is similar to the one of Unix's yacc. During syntax analysis when a syntax error is detected the analyser switches to recovery mode in order to continue further analysis to detect the remaining errors. In our implementation we do not use recovery mode. Instead, we focus on reporting an error in the best possible way. We augment the grammar for the language with productions that generate erroneous constructs. The Python yacc provides a special token `error` that acts as a wildcard for any erroneous input. An analyser constructed from a grammar augmented by these error productions is useful for detecting anticipated errors.

#### Symbol table

Symbol tables are data structures used by compilers to hold information about identifiers coalesced from the program's source code. A semantic action puts information about identifier $x$ into the symbol table, when the declaration of $x$ is analysed, and uses it when necessary.

The scope of a declaration is the portion of the program to which the declaration applies. In synchroniser code state and store variables are visible for all states and transitions. State expression aliases and pattern-matched variables are visible only within the current transition. We have implemented scopes using a chained symbol table approach described in [17]. We set up a root symbol table for state and store variables and separate symbol tables for each transition. The tables are chained as illustrated in Fig. 4.14.

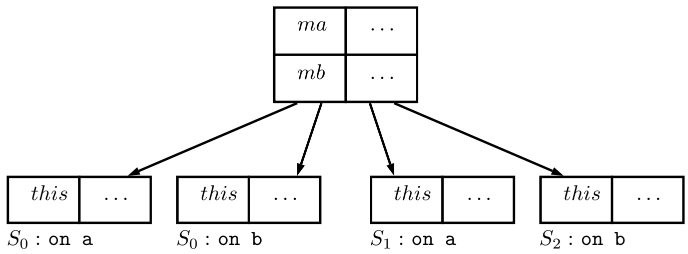


FIGURE 4.14: Scoping structure of a synchroniser program using the zip2 synchroniser as an example

The symbol table implementation supports three operations: create new symbol table, put new entry in table and get entry for identifier from table. In the sequel, we refer to these operations as *NewSymtab*(symtab), symtab.*put*(*ID, value*) and symtab.*get*(*ID*). The pseudo-code implementations of symtab.*get* and symtab.*put* are given in Fig. 4.15 and Fig. 4.16 respectively.

The synchroniser language does not permit using of the reserved words (Appendix A.2) as identifiers. This is checked before putting an identifier in the symbol table.

### 4.4.3 Static Analysis

Static analysis includes:

- Semantic checking. Constraints such as an identifier is declared at most once in a scope

```
function GET(symtab, id)
    while symtab is not nil do
        tmp ← get(symtab, id)
        if tmp then
            return tmp
        else
            symtab ← previous(symtab) ▷ previous(symtab) returns a symbol table
of the most-closely outer scope
        end if
    end while
    return not_found
end function
```

FIGURE 4.15: Getting an entry for an identifier from the chained symbol table

```
function PUT(symtab, id, value)
    if id is reserved then
        error
    end if
    tmp ← get(symtab, id)
    if not tmp then
        symtab ← (id, value)
    else
        error
    end if
end function
```

FIGURE 4.16: Putting a new entry in the symbol table

- Type checking. The type rules of a language assure that an operator or function is applied to the right number and type of operands.

### Semantic checking

In this section we describe the semantic checks that are not enforced by the grammar.

The synchroniser language requires identifiers except for state expression aliases to be declared before they are used. Moreover, an identifier must be declared at most once in a scope. The channels, on which the transitions in the synchroniser are made, must be declared in the channel signature as well as the channels where messages are sent. In order to check those, we maintain three symbol tables for identifiers, input channels and output channels. The scheme given in Fig. 4.1 shows how the symbol tables are managed and used during the syntax analysis. The symbol tables are initialised before the syntax analyser runs, as shown in Fig. 4.17.

The attribute *type* is added to each entry in the symbol table. State variables are assigned type *integer*. Non-integer variables, whose structure is unknown, are assigned a special type *void*, which stands for a variable MDL term. A detailed information about the synchroniser language type checker is provided in section 4.4.3.

```
function INIT
    InChantab ← NewSymtab(nil)
    OutChantab ← NewSymtab(nil)
    RootSymtab ← NewSymtab(nil)
end function
```

FIGURE 4.17: Initialisation of symbol tables

The synchroniser compiler checks if the transition diagram of a synchroniser is connected. The algorithm given in Fig. 4.18 walks the abstract syntax tree *synch* and constructs

<table>
<tr><th>Production</th><th>Semantic Action</th></tr>
<tr><td>⟨synch⟩ ::= ‘synch’ ⟨ID⟩ ‘(’ ⟨input⟩ [‘,’ ⟨input⟩]*<br>‘|’ ⟨output⟩ [‘,’ ⟨output⟩]* ‘)’ ‘{’ ⟨decl⟩* ⟨state⟩+ ‘}’</td><td>/* synch is the abstract syntax tree of the source code. */</td></tr>
<tr><td>⟨input⟩ ::= ⟨chan⟩ [‘:’ (⟨ID⟩ | ⟨NUMBER⟩)]<br>⟨chan⟩ ::= ⟨ID⟩</td><td>foreach ⟨chan⟩<br>InChantab.put(⟨chan⟩, type=void)</td></tr>
<tr><td>⟨output⟩ ::= ⟨chan⟩ [‘:’ ⟨depth_exp⟩]<br>⟨chan⟩ ::= ⟨ID⟩</td><td>foreach ⟨chan⟩<br>OutChantab.put(⟨chan⟩, type=void)</td></tr>
<tr><td>⟨decl⟩ ::= ‘store’ ⟨store_id_list⟩ ‘;’<br>| ‘state’ ⟨type⟩ ⟨state_id_list⟩ ‘;’<br>⟨store_id_list⟩ ::= ⟨id_list⟩<br>⟨state_id_list⟩ ::= ⟨id_list⟩</td><td>foreach id in ⟨store_id_list⟩<br>Symtab.put(id, type=void)<br>foreach id in ⟨state_id_list⟩<br>Symtab.put(id, type=int)</td></tr>
<tr><td>⟨trans_stmt⟩ ::=<br>⟨trans_name⟩ [‘.’ ⟨condition⟩] [‘&’ ⟨int_exp⟩ ] ⟨actions⟩</td><td></td></tr>
<tr><td>⟨trans_name⟩ ::= ⟨ID⟩</td><td>if not InChantab.get(⟨ID⟩)<br>error<br>Symtab = NewSymtab(RootSymtab)<br>Symtab.put(‘this’, type=void)</td></tr>
<tr><td>⟨condition⟩ ::= ‘@’ ⟨segmark⟩<br>| ‘?’ ⟨ID⟩<br>| ‘else’<br>⟨segmark⟩ ::= ⟨ID⟩</td><td>if not Symtab.get(⟨segmark⟩)<br>error</td></tr>
<tr><td>⟨condition⟩ ::= [‘?’ ⟨ID⟩] ‘(’ ⟨id_list⟩ [‘||’ ⟨tail⟩ ]‘)’</td><td>foreach id in ⟨id_list⟩ ∩ ⟨tail⟩<br>Symtab.put(id, type=void)<br>tmp = Symtab.get(‘this’)<br>tmp.type = { ‘$p_1$’:void, ...‘$p_n$’:void }, where $p_1$, ...$p_n$ are elements of ⟨id_list⟩<br>if ⟨tail⟩<br>tmp.type = tmp.type | ⟨tail⟩</td></tr>
<tr><td>⟨send_stmt⟩ ::= ‘send’ ⟨dispatch⟩ [‘,’ ⟨dispatch⟩* ‘;’<br>⟨dispatch⟩ ::= ⟨msg_exp⟩ ‘=>’ ⟨ID⟩</td><td>tmp = OutChantab.get(⟨ID⟩)<br>if not tmp<br>error</td></tr>
</table>

TABLE 4.1: Symbol tables management and duplicate declaration checking scheme

```
function GET_STATES(synch)
    StateSet ← nil
    GotoSet ← nil
    for each state in state_list(sync) do
        if label(state) ∈ StateSet then
            error                                   ▷ if the state label is not unique
        else
            StateSet ← StateSet.Append(label(state))
        end if

        for each trans in trans_list(state) do
            for each gotostate in goto_list(trans) do
                if gotostate ∉ GotoSet then
                    GotoSet ← GotoSet.Append(gotostate)
                end if
            end for
        end for
    end for

    return (StateSet, GotoSet)
end function
```

FIGURE 4.18: Construction of *StateSet* and *GotoSet*

| Type | Values |
|---|---|
| $int(n)$ | $[0; 2^n - 1]$ |
| $enum(a_1, a_2, \ldots\ a_n)$ | $[0; n-1]$ |
| $enum(a_1 = N_1, a_2 = N_2, \ldots\ a_n = N_n)$ | $N_1, N_2, \ldots\ N_n$ |

FIGURE 4.19: Computing the value range of an integer variable

two sets: the set of the synchroniser state labels *StateSet* and the set of the goto labels *GotoSet*. The set $GotoSet \setminus StateSet$ contains goto labels that point to non-existent states. If this set is not empty the compilation reporting an error. The set $StateSet \setminus (GotoSet \cup$ `'start'`$)$ contains unreachable states that are eliminated, also reporting an error. The algorithm also checks if the labels of the synchroniser states are unique.

A synchroniser program must have the state labeled `'start'`. The existence of this state is checked once the *StateSet* is constructed.

The synchroniser language provides two types of state variable: an integer of the specified size and an enumeration. The size defines the value range of the integer. The value range of the enumeration is specified in the variable declaration. Fig. 4.19 gives formulae for the value range computation.

For a state expression that evaluates to an integer we can check if the computed value belongs to the assignment-destination value-range. The symbol table stores the value-range information for integer entries and provides an interface to it in the form of boolean function `check_range`(*id*, *value*). The function returns *true* if the *value* fits in the *id* value range and *false* otherwise. Fig. 4.21 shows how the check is integrated into the syntax analyser.

**Tests** We have developed a test suite for the semantic analyser. It uses the standard python unit testing framework *unittest*. The tests expand the abstract syntax tree into a nested list and compare the result with the expected value.

### Type checking

The design of the type checker is based on information about the syntactic constructs in the language, the notion of types and the rules for assigning types to language constructs. The type of a language construct is denoted by a type expression. Informally, a type expression is either a basic type or the application of an operator called a type constructor to other type expressions. A collection of rules for assigning type expressions to language constructs is called a type system.

The communication protocol of the AstraKahn runtime system prototype supports only choices. When a synchroniser reads a message from its input channel, the record that belongs to the variant specified in the synchroniser transition is instantiated[2]. We take this into account in implementing the type checker of the synchroniser language. Because the synchronisers can read values only of the fields that are known to be integer, the basic types in the synchroniser language are integer and MDL variable. Integer is the type of state variable. MDL variables are building blocks for the only constructed type in the synchroniser language – a record. A record is constructed by the concatenation of two records. The pseudo-code of the record constructor || is given in Fig. 4.20. The record constructor is obviously commutative and associative.

The case when a label-value pair labeled $l$ exists in both operand records $r_1$ and $r_2$ is indicated with $union(r_1(l), r_2(l))$. The constraint solver resolves which option to take during the constraint aggregation pass in the AstraKahn compiler.

We describe the type checkers in terms of grammar productions and corresponding semantic actions. The type checkers for state and store expressions are given in Fig. 4.22 and Fig. 4.23 respectively. The synthesized attribute *type* for an expression $\langle E \rangle$ gives the type of the expression assigned by the type system for the expression generated

[2] this means that store variables cannot keep multiple variants

```
function ||(r1, r2)
    if len(r1) ≤ len(r2) then
        r_iter ← r1
        r ← r2
    else
        r_iter ← r2
        r ← r1
    end if                    ▷ r is the record that contains fewer label-value pairs
    for each label-value pair (l,v) in r_iter do
        if r(l) then                                    ▷ if label l exists in r
            r(l) ← union(r(l), v)
        else
            r(l) ← v
        end if
    end for
    return r
end function
```

FIGURE 4.20: The record type constructor ||

| Production | Semantic Action |
|---|---|
| ⟨*assign*⟩ ::= ⟨*dest*⟩ '=' '[' ⟨*int_exp_c*⟩ ']'<br>⟨*dest*⟩ ::= ⟨*ID*⟩ | tmp = Symtab.*get*(⟨*dest*⟩)<br>*if not* tmp<br>Symtab.*put*(⟨*dest*⟩, *type*=*int*)<br>*else*<br>*if* tmp.*type* != *int*<br>*error*<br>*if* ⟨*int_exp_c*⟩ *evaluates to int*<br>*if not check_range*(⟨*dest*⟩, *eval*(⟨*int_exp_c*⟩))<br>*error* |
| ⟨*assign*⟩ ::= ⟨*dest*⟩ '=' ⟨*data_exp*⟩<br>⟨*data_exp*⟩ ::=<br>(⟨*data*⟩ \| '(' ⟨*data*⟩ ')') | tmp = Symtab.*get*(⟨*dest*⟩)<br>*if not* tmp<br>*error*<br>*if* tmp.*type* == *int*<br>*error*<br>tmp.*type* = ⟨*data_exp*⟩.*type* |

FIGURE 4.21: Type checker for statements

by ⟨*E*⟩. The type checker for statements is given in Fig. 4.21. It assures that the left hand side can be assigned to.

<table>
<tr><th>Production</th><th>Semantic Action</th></tr>
<tr><td>⟨int_exp_c⟩ ::= ⟨NUMBER⟩ | ⟨ID⟩</td><td>tmp = Symtab.get(⟨ID⟩)<br>if not tmp<br>  error<br>if tmp.type != int<br>  error<br>⟨int_exp_c⟩.type = int</td></tr>
<tr><td>⟨int_exp_c⟩ ::= ‘(’ ⟨int_exp_c$_1$⟩ ‘)’<br>  | ‘-’ ⟨int_exp_c$_1$⟩<br>  | ‘!’ ⟨int_exp_c$_1$⟩</td><td>⟨int_exp_c⟩.type = ⟨int_exp_c$_1$⟩.type</td></tr>
<tr><td>⟨int_exp_c⟩ ::=<br>    ⟨int_exp_c$_1$⟩ ⟨op⟩ ⟨int_exp_c$_2$⟩<br>⟨op⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’<br>  | ‘<<’ | ‘>>’ | ‘|’ | ‘&’ | ‘^’<br>  | ‘<’ | ‘>’ | ‘==’ | ‘!=’ |‘<=’<br>  | ‘>=’ | ‘&&’ | ‘||’</td><td>if ⟨int_exp_c$_1$⟩.type == int<br>  and ⟨int_exp_c$_2$⟩.type == int<br>  ⟨int_exp_c⟩.type = int<br>else<br>  error</td></tr>
</table>

FIGURE 4.22: Type checker for state expressions

<table>
<tr><th>Production</th><th>Semantic Action</th></tr>
<tr><td>⟨data⟩ ::= ⟨item_list⟩<br>⟨item_list⟩ ::= ⟨item⟩ [‘||’ ⟨item⟩]*</td><td>foreach item in ⟨item_list⟩<br>  ⟨data⟩.type = ⟨data⟩.type || item.type</td></tr>
<tr><td>⟨item⟩ ::= ‘this’</td><td>tmp = Symtab.get(‘this’)<br>⟨item⟩.type = tmp.type</td></tr>
<tr><td>⟨item⟩ ::= ⟨ID⟩</td><td>tmp = Symtab.get(⟨ID⟩)<br>if not tmp<br>  error<br>if tmp.type == int<br>  error<br>⟨item⟩.type = tmp.type</td></tr>
<tr><td>⟨item⟩ ::= ‘’’ ⟨ID⟩</td><td>tmp = Symtab.get(⟨ID⟩)<br>if not tmp<br>  error<br>⟨item⟩.type = { ‘ID’:tmp.type }</td></tr>
<tr><td>⟨item⟩ ::= ⟨ID⟩ ‘:’ ⟨rhs⟩<br>⟨rhs⟩ ::= ⟨int_exp⟩ | ⟨rhs_ID⟩<br>⟨rhs_ID⟩ ::= ⟨ID⟩</td><td>if ⟨rhs_ID⟩<br>  tmp = Symtab.get(⟨rhs_ID⟩)<br>  if not tmp<br>    error<br>tmp = Symtab.get(⟨ID⟩)<br>⟨item⟩.type = { ‘ID’:⟨rhs⟩.type }</td></tr>
</table>

FIGURE 4.23: Type checker for store expressions

### 4.4.4 Code Generation

#### Synchroniser runtime code

The compiler generates an abstract syntax tree (AST) that is passed to the AstraKahn runtime system for the interpretation. An example of the AST for the counter synchroniser in Fig. 4.24 is shown in Fig. 4.24. A detailed description of the AST structure can be found in Appendix A.3.

```
1  synch counter (a | c)
2  {
3    state int(8) i = 0;
4    start {
5      on:
6        a & [i < 255] {
7          set i = [i + 1];
8        }
9        a && [i = 255] {
10         set i = [0];
11         send this => c;
12       }
13   }
14 }
```

FIGURE 4.24: The counter synchroniser code

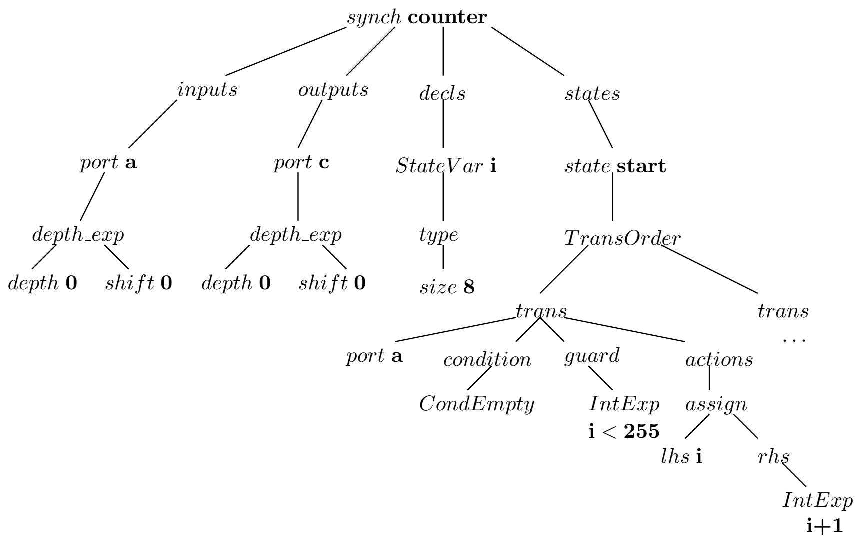


FIGURE 4.25: The AST generated for the counter synchroniser in Fig. 4.24. The AST is drawn only for the transition in lines 6-8. The AST generated for the transition in lines 9-12 is similar

#### Synchroniser passport

A synchroniser passport consists of the MDL terms for its input and output channels. Those terms are not straightforward since they have to be fairly generic to match the broadest possible formats of producer and consumer messages involved in the act of synchronisation. On the other hand, the synchroniser passport is produced solely on the basis of the synchroniser code, exclusively by program analysis; the programmer does not supply an explicit passport for this.

The first thing that is required is the input interface. As mentioned above, the communication protocol of the Astra*Kahn* runtime system prototype supports only choices. The use of a variant on a channel in any transition on that channel immediately associates the *choice* term structure with it. For example, a channel $a$ that is tested on variants $?v$ and $?w$ in transitions has a term comparable with **(:** `'v'`:*$vterm*, `'w'`:*$wterm* || *$tail* **:)**, where the variables *$vterm* and *$wterm* represent the terms for variants $v$ and $w$ and *$tail* represents the choice term that contains the rest of the variants. If a choice is known to carry a single variant, the variant is labeled *uniq*. A transition that does not specify a variant label expects a unique message variant.

The symbol table for a transition maintains a special entry `'this'`. It holds the term of the message accepted by the transition. The algorithm in Fig. 4.26 walks the synchroniser transitions and constructs the input term for the synchroniser[3].

Since a choice is in fact a collection of label-record pairs, the constructor || (Fig. 4.20) can be applied for choices. When *InputTerm* contains a label-value pair (`'v'`, *value*) and the label `'v'` occurs in another transition, the union || of these two choices results in the following:

$$\begin{aligned} & \mathit{union}\,((\mathbf{:}\ \texttt{'v'} : \$\mathit{vterm}_1 \mid \$\mathit{tail}_1\ \mathbf{:}),\ (\mathbf{:}\ \texttt{'v'} : \$\mathit{vterm}_2 \mid \$\mathit{tail}_2\ \mathbf{:})) \\ = \quad & (\mathbf{:}\ \texttt{'v'} : \mathit{union}\,(\$\mathit{vterm}_1,\ \$\mathit{vterm}_2) \mid (\$\mathit{tail}_1 \mid\mid \$\mathit{tail}_2)\ \mathbf{:}) \end{aligned}$$

Now consider the output interface. The algorithm in Fig. 4.27 collects the dispatches for every output channel and combines them into the output term of every output channel.

The function *type*(*id*) performs the symbol table lookup and returns the *type* attribute value for the `'id'` entry.

## 4.5 Discussion and Future Work

In this chapter we have presented the language for Astra*Kahn* synchronisers. We have developed the language compiler that generates the passport of the synchroniser. The message format in Astra*Kahn* is based on the Message Definition Language (MDL) with

[3] The algorithm relies on the abstract syntax tree structure (see Appendix. A.3)

```
function INPUT_TERM(synch)
    CondDict ← nil

    for each state in state_list(sync) do ▷ Construct a dictionary CondDict with
  input ports as keys and sets of corresponding message conditions as values
        for each trans in trans_list(state) do
            cond ← get_condition(trans)
            if port ∈ CondDict then
                CondDict(port) ← CondDict(port).Append(cond)
            else
                CondDict ← CondDict.Append((port, cond))
            end if
        end for
    end for

    InputTerm ← nil
    for each (port, cond_set) in CondDict do
        PortTerm ← nil
        for each cond in cond_set do
            variant ← get_variant(cond)
            this ← Symtab.get('this')
            if variant is unique then
                if PortTerm ≠ nil then error
                end if
                PortTerm ← (: 'uniq':this :)
                break
            else
                PortTerm ← PortTerm || this
            end if
        end for
        InputTerm ← InputTerm.Append((port, PortTerm))
    end for

    return InputTerm || $tail
end function
```

FIGURE 4.26: Construction of the synchroniser input term

the restriction that data on streams are organised as collections of alternative records of label-value pairs. A value in a record can have any structure that is allowed by the MDL. The synchroniser that we have presented matches only a top-level structure of a message. The MDL generates a much broader set of terms than the current version synchroniser can synchronise; however, it needs to be elaborated whether the implementation of lower level synchronisation in synchronisers is useful for the real world applications.

The current version of the language does not define flow inheritance in synchronisers. The synchroniser *code* only needs to access the label-value parts of the message it

```
function OUTPUT_TERM(synch)
    DispatchDict ← nil
    for each state in state_list(sync) do   ▷ Construct a dictionary DispatchDict
  with output ports as keys and sets of corresponding messages as values
        for each trans in trans_list(state) do
            send ← get_send(trans)
            (msg, port) ← (get_msg(send), get_port(send)
            if port ∈ DispatchDict then
                DispatchDict(port) ← DispatchDict(port).Append(msg)
            else
                DispatchDict ← DispatchDict.Append((port, msg))
            end if
        end for
    end for

    OutputTerm ← nil
    for each (port,msg_set) in DispatchDict do
        PortTerm ← nil
        for each msg in msg_set do
            if msg is MsgData then                ▷ msg matches [‘?’⟨ID⟩]⟨data⟩
                (variant, data) ← (get_variant(msg), get_data(msg))
                for each item in data do
                    if item is ItemVar or item is ItemPair then ▷ item is either ’id
  or id:value
                        (lhs, rhs) ← expand(item)
                        PortTerm ← PortTerm || {lhs:type(rhs)}
                    else                          ▷ item is either this or id
                        PortTerm ← PortTerm || type(id)
                    end if
                end for
            end if
        end for
        OutputTerm ← OutputTerm.Append((port, PortTerm))
    end for

    return OutputTerm || $tail
end function
```

FIGURE 4.27: Construction of the synchroniser output term

matches. Thus, the flow inheritance should be supported outside of the synchroniser definition and the question how it should be done is left open.

The compiler we have implemented does not optimise the code deeply. We have only implemented an elimination of unused states. However, a broader set of dead code elimination optimisations can be implemented[4].

[4]E.g. a conditional transition can be eliminated if its condition always evaluates to *false*

# Chapter 5

# Serial Replication in AstraKahn

In this chapter we explain the machinery behind the serial replication in AstraKahn and the role of synchronisers in it. We introduce the concept of the forward fixed point for the replication pipeline and show how it is used to organise the output from the infinite chain of replicas. In order to suppress the growth of the replica chain, we present the concept of the reverse fixed point and show how it is used to optimise some replicas at the head of the chain.

## 5.1 AstraKahn Approach to Serial Replication

The serial replication combinator creates a conceptually infinite number of copies of its operand network, and connects them in a chain. Replication is demand-driven: the replicas are created dynamically. A fresh replica is *inactive*[1], hence it does not necessarily require significant resources since AstraKahn boxes are stateless and since synchronisers require no resources in their start state[2]. Indeed the cost of replication is only felt when the replicas are active, which is the case from the time that the first message is received until all messages have left the replica and all its synchronisers have returned to their start states.

In S-NET, the output from a replication pipeline is based on the record subtyping in the type system. The replication combinators in S-NET require the programmer to specify a termination pattern, so that each record that is a subtype of this pattern leaves the replication pipeline throught the output stream.

[1]More generally, we call a replica inactive when all of its synchronisers are in their start states, none of its channels has messages in them and no box is running

[2]When a synchroniser transitions back to the start state, it flushes its store variables

In Astra*Kahn* the output from the replication pipeline is defined using the concept of fixed point. From the mathematical point of view a fixed point of a function is an element of the function's domain that it maps to itself. That is to say, a function $f(x) : X \to Y$ has a fixed point at $x_0 \in X$ if $f(x_0) = x_0$. The serial replication combinator implements the computation[3] shown in Fig. 5.1. After the $n$-th replica has processed the message $f^{(n-1)}(x) \neq x_0$ the computation reaches the fixed point $f^{(n)}(x) = x_0$, and the message $x_0$ is sent on to the output channel of the serial replication network $f^*$.

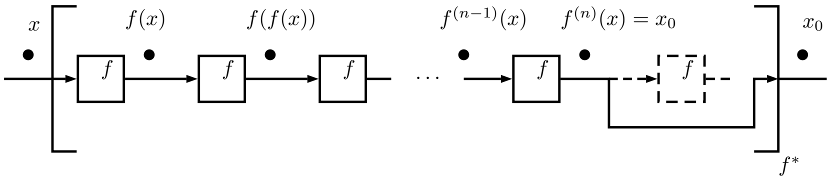


FIGURE 5.1: The recursive computation in Astra*Kahn*

The number of iterations $n$ needed to reach the fixed point is not known in advance, meaning that in order to utilize the fixed point as shown in Fig. 5.1, Astra*Kahn* must be able to detect the fixed point message right at the time it is produced by the $n$-th replica or later. Therefore, similar to S-NET, Astra*Kahn* needs to be provided with a pattern that matches all the fixed point messages of the operand network. In S-NET, the serial replication combinator requires this pattern as one of its operands; by contrast, in Astra*Kahn* the pattern is required to be deduced from the operand network by compiler analysis.

The chain of replicas grows as the computation progresses, however, in the example in Fig. 5.1 the computation is carried out only by a single replica in the tail of the chain. The replicas in the head of the chain have processed the message and are not used anymore. In order to suppress the growth of the chain, Astra*Kahn* must detect such replicas and optimise the connection by removing them. Since Astra*Kahn* boxes are stateless, an operand network can have a state that is fully defined by the states of its synchronisers. A replica of the operand network can be removed from the chain safely if the replica is in a state, in which it forwards any message without change, i.e. any message it receives is its fixed point. We will call such a state of the replica a reverse fixed point state.

## Motivating Example

In this section we demonstrate the concept of fixed point by a simple example.

[3] $f^{(n)}(x)$ denotes $\underbrace{f(f(\ldots f}_{n}(x)\ldots))$

The serial replication network $f^*$ in Fig. 5.2 computes a total sum of integer values that are stored under label $x$ in incoming messages while the sum is less than some integer value $n$. The network has one input and one output channel $x$. Although the synchronisation can be done by variants, for the sake of simplicity we consider that the input message has a unique variant and, thus, the message format of the operand network $f$ is (: **uniq**: {**x**:*Int*} :). In order to compute a sum, the value stored under label $x$ has to be extracted out of two input messages, so the network has to preserve a state by keeping the message that was received first. Once the second message is received, the network can proceed to a stateless computation and finally send the result on to the next replica of $f$.

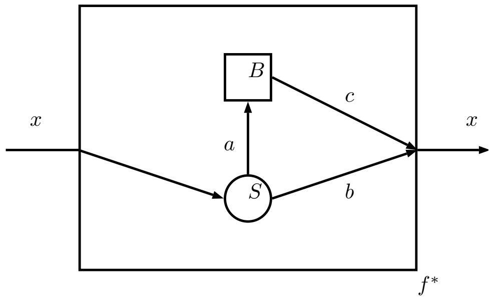


FIGURE 5.2: A motivating example for forward and reverse fixed points

The state management in $f$ is implemented by synchroniser $S$ listed in Fig. 5.3. The synchroniser has one input channel $x$, which is also the input channel of $f$, and two output channels $a$ and $b$. Channel $a$ is connected to a box ($B$) that performs the summation, and channel $b$ merges with the output channel of $B$ into the output channel of $f$. The synchroniser declares a store variable ($ma$) for keeping the first summand. In the start state the synchroniser checks the integer value stored under label $x$ in the received message. If that value is greater than $n$, the message is sent on to the output channel $b$ and then to the next replica of $f$. Otherwise, the label-value pair **x**:*value* is saved to the store variable and synchroniser $S$ transitions to state $s1$. In state $s1$ the synchroniser extracts the integer value from the received message and forms an output message of format (: **uniq**: {**x**:*Int*, **y**:*Int*} :) using the contents of the store variable and the extracted value. Then the message is sent on to $B$ and the synchroniser transitions to state $rfp$ (**r**everse **f**ixed **p**oint state).

Box $B$ sums up the values stored under labels $x$ and $y$ of the received message and forms an output message (: **uniq**: {**x**:*Int*} :) carrying the sum of the two values as shown in Fig. 5.4. The message is sent on to the output channel of $B$ and consequently to the output channel of $f$.

In state $rfp$ synchroniser $S$ transmits any message it receives on its input channel without change and then transitions back to this state. When $S$ stays in state $rfp$, the

```
1  synch S (x | a, b)
2  {
3    store ma:x;
4    start {
5      on:
6        a.(x || t) & [x >= n] {
7          send this => b;
8        }
9        a.(x || t) & [x < n] {
10         set ma = x:[x];
11         goto s1;
12       }
13   }
14   s1 {
15     on:
16       a.(x || t) {
17         send ma || y:[x] => a;
18         goto rfp;
19       }
20   }
21   rfp {
22     on:
23       a {
24         send this => b;
25         goto rfp;
26       }
27   }
28 }
```

FIGURE 5.3: Synchroniser S in Fig. 5.2

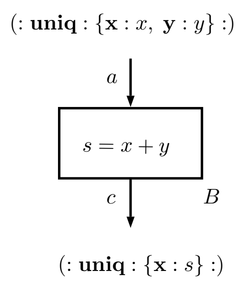


FIGURE 5.4: Schematic box code for the motivating example in Fig. 5.2

whole replica of $f$ acts in the same way. Such replicas can be removed from the wiring in order to unlock the reserved resources. The removal of these replicas is necessary because the number of replicas needed for the computation is not known in advance. When the total sum is greater than $n$ and the summation is finished, the next summation is initiated in $f^*$ with the arrival of a new message. If this summation uses less replicas than the previous one, the extra replicas should be noted and removed to release the resources. The behaviour that is coded in state $rfp$ of synchroniser $S$ allows the Astra*Kahn* runtime to detect unused replicas and remove them.

The transition in line 6 of Fig. 5.3 checks if the interim result of the summation is greater than $n$. If the condition is satisfied, the received message is sent on to the next replica of $f$. Synchroniser $S$ in that replica receives the message and the same transition is activated resulting in the message being sent on to the next replica. In this way the message that carries the interim result greater than $n$ cascades through the infinite chain of replicas without change. Thus, the message is the fixed point of the serial replication network $f^*$. Generally, a fixed point message of $f^*$ is detected by testing the condition $(x \mid\mid t)$ & $x \geq n$ on the message, and this test is encoded in the transition in line 6 of $S$ (Fig. 5.3). In order to detect fixed point messages, program analysis extracts the condition from the synchroniser source code. Once the fixed point condition is known, Astra*Kahn* is able to detect fixed point messages on runtime.

In the remainder of the chapter we will give formal definitions of a fixed point message and a reverse fixed point state, and will provide algorithms for the Astra*Kahn* compiler to detect them. In the sequel, we will call a fixed point message a forward fixed point.

## 5.2 Forward Fixed Point

Once the computation in a serial replication network has reached the fixed point, newly created replicas are known to transmit fixed point messages without change. Astra*Kahn* does not analyse boxes[4], so it can determine about the operand network behaviour only from its synchronisers. Thus, in order for the operand network to be analysable by Astra*Kahn* it must contain a path that does not traverse boxes and which may traverse synchronisers. Because a newly created replica is inactive, and hence the synchronisers in it are in their start states, the start states of the synchronisers that belong to the path must have a special transition that sends the message on to the next synchroniser along the path. Since transitions can be conditional on the message content, the fixed point pattern, or rather the fixed point condition, can be present in these special transitions.

The existence of a forward fixed point requires the operand network to have some topological properties that are formally defined as follows. Consider a network $N$ that has an input and an output channel, both named $x$.

**Definition 1.** The network $N$ is said to have a forward fixed point in $x$ if and only if the following requirements are satisfied:

1. There exists a condition $P(m)$ on the content of the message $m$ received by the network on the input channel $x$ under which it follows a unique non-branching path to the output channel $x$ without traversing any boxes
2. The path[5] can traverse synchronisers, but then whenever $P(m)$ is true and the synchroniser is in the start state, it must accept $m$ and transition back to the start

[4] Except for the communication passport generation
[5] In the sequel, we will call this path the fixed point path

state while sending the message $m$ on the path unchanged and without producing any other output

The condition $P$ may not be unique for each network, and when it is not, the fixed point condition of the network is a disjunction of all such conditions. The condition can also be a tautology, in which case the forward fixed point is called unconditional. When the fixed point path traverses a single synchroniser, the fixed point condition is defined exclusively by the synchroniser transitions that loop around the start state and send on the accepted messages unchanged. When the path traverses several synchronisers, the fixed point condition of the network is a conjunction of the fixed point conditions of these synchronisers. We demonstrate the construction of the fixed point condition with the example operand network $N$ depicted in Fig. 5.5.

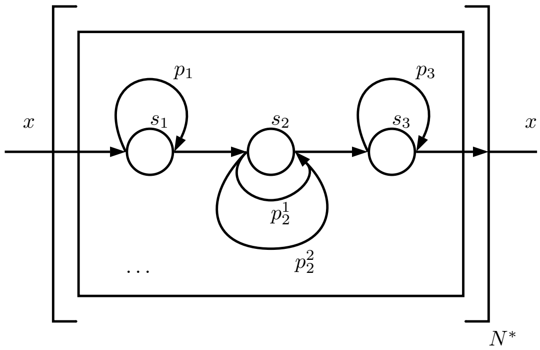


FIGURE 5.5: Forming of the forward fixed point condition of a network

Apart from all the paths that traverse boxes, the operand network $N$ has a unique path $[s_1,\ s_2,\ s_3]$ that traverses only synchronisers. The synchroniser $s_2$ has two transitions that loop around the start state and send on messages they accept unchanged with the firing conditions $p_2^1$ and $p_2^2$. A message $m$ is a fixed point for the synchroniser $s_2$ when it satisfies any of these conditions, i.e. $p_2^1(m) \vee p_2^2(m)$ is true. The synchronisers $s_1$ and $s_3$ have fixed point conditions $p_1$ and $p_3$ respectively. Then the fixed point condition of the network $N$ is $p_1 \wedge (p_2^1 \vee p_2^2) \wedge p_3$.

Definition 1 requires the synchronisers that traverse the fixed point path to transition back to the start state after the fixed point message has been sent to the output channel. This restriction can be relaxed, however, in this case the programmer would have to maintain transitions that check the fixed point condition in every state of each synchroniser along the path.

### Output from the Serial Replication Network

Now we will clarify how the serial replication network is wired to the rest of the AstraKahn application network and how the output is produced. Strictly speaking, the serial replication is not just a wiring pattern since it does not simply wire the replicas of its operand network. It also creates a set of output channels and augments the replicas with some auxiliary synchronisers.

The serial replication $N^*$ defines the output channel set $\mathcal{N}_{out}$ as follows:

$$\mathcal{N}_{out} = \{name(c) \mid c \in \mathcal{O} \wedge fp(c)\}$$

where $\mathcal{O}$ is the output channel set of $N$ and the predicate $fp(c)$ is true on any channel $c$ that has a forward fixed point. The serial replication creates a set of fresh output channels $\mathcal{O}^*$ taking the names from the set $\mathcal{N}_{out}$. A message that is sent to an inactive replica on any channel $c$ with $name(c) \in \mathcal{N}_{out}$ and which satisfies the fixed point condition on that channel is immediately transferred to the identically named output channel from $\mathcal{O}^*$. A network in Fig. 5.6 demonstrates how the output is produced from the serial replication of a network that has a single input and a single output channel.

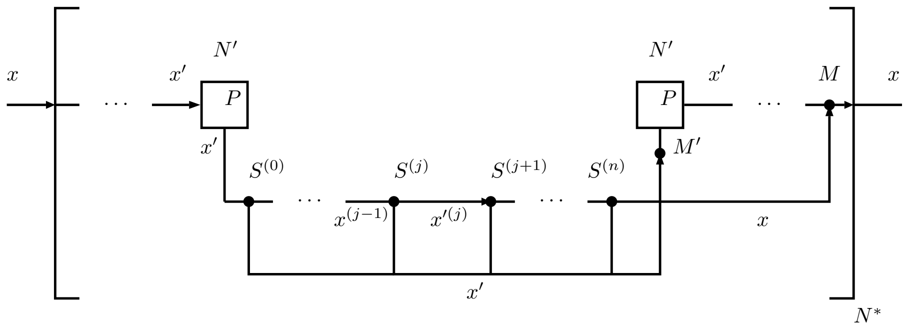


FIGURE 5.6: Output from the serial replication network

The operand network $N$ has the fixed point condition $P = \bigvee_{j=0}^{n} p_j$, where $p_j$ is the fixed point condition extracted from the $j$-th synchroniser ($0 \leq j \leq n$) on the fixed point path of $N$. In order to check whether a message $m$ on channel $x'$ satisfies the condition $p_j$, a synchroniser $S^{(j)}$ is inserted before every inactive replica $N'$. If $p_j(m)$ is true, the synchroniser sends the message $m$ on to the input channel of the next synchroniser $S^{(j+1)}$ to check whether $p_{j+1}$ is satisfied. Otherwise, the message $m$ is sent to the input channel $x'$ of the next replica of $N$. The listing of the synchroniser $S^{(j)}$, $1 \leq j \leq n-1$ is given in Fig. 5.7. The synchronisers $S^{(0)}$ and $S^{(n)}$ have the same structure, however, $S^{(0)}$ reads messages from the output channel $x'$ of the previous replica and $S^{(n)}$ sends the message that satisfies $P$ to the output of $N^*$. We shall note that for all $j$ for which $p_j = \bigwedge_{i=0}^{k} p_j^{(i)}$ the synchroniser $S^{(j)}$ has $k$ transitions that check the conditions $p_j^{(i)}$.

```
synch S^(j) (x'^(j-1) | x'^(j), x')
{
  start {
    on:
      x'^(j-1).p_j {
        send this => x'^(j);
      }
      x.else {
        send this => x';
      }
  }
}
```

FIGURE 5.7: The synchroniser $S^{(j)}$

The merger $M'$ gathers messages that do not satisfy the fixed point condition from all synchronisers $S^{(j)}$ between two consecutive replicas of $N$ and forwards them to the input channel of the next replica. The merger $M$ gathers messages that satisfy the fixed point condition $P$ and forwards the messages to the output channel $x$ of the serial replication network $N^*$.

### 5.2.1 Forward Fixed Point Detection

The existence of a forward fixed point requires the synchronisers that are traversed by the fixed point path to have at least one transition in their start states that accepts the fixed point messages and sends them on unchanged and without producing any other output. Coded in the synchroniser language, the transition that defines the fixed point condition $p$ in channel $x$ is presented in Fig. 5.8.

```
start {
  on:
    x.p {
      send this => out;
    }
  ...
}
```

FIGURE 5.8: The start state of a synchroniser that encodes the fixed point condition $p$ in channel $x$

Note that there exists no other transition on the channel from the start state with the same structure (a single *send* clause). Otherwise, the synchroniser would have the fixed point condition that is the disjunction of conditions in such transitions. The algorithm in Fig. 5.9 checks if a forward fixed point exists on channel $x$ and extracts the fixed point condition from the synchroniser source code. The algorithm supports the renaming of channel $x$ in the synchroniser. Moreover, it may be the case that the transitions that cause a fixed point in the synchronisers send messages to different output channels. The algorithm detects such a situation; however, the branching of the fixed point path is resolved in the context of the whole network.

**Require:** the abstract syntax tree of the synchroniser program (*synch*), the input label of a channel to test for a forward fixed point ($x$)
**Ensure:** a dictionary ($a, CondList$), where $a$ is the output label of the fixed point channel and $CondList$ is the list of atomic fixed point conditions

```
function EXTRACT_FP(synch, x)
    state ← getthestartstatetreefromsynch
    CondDict ← nil

    for each trans in trans_list(state) do
        if get_port(trans) ≠ x then
            continue                ▷ the transition reads from another channel
        end if
        if get_goto(trans) ≠ ('start' ∨ ∅) then
            continue                ▷ the transition does not loop around the state state
        end if
        if get_assign(trans) ≠ ∅ then
            continue
        end if
        send ← get_send(trans)
        if get_msg(send) ≠ this then
            continue
        end if
        cond ← get_condition(trans)
        if cond is not CondDataMsg ∨ cond is not CondEmpty then
            continue    ▷ the condition cannot be a segmentation mark or an .else
        end if
        out_port ← get_port(send)
        if cond ∈ CondDict(out_port) then
            CondDict(out_port)← CondDict(out_port).Append(cond)
        else
            CondDict ← CondDict.Append(out_port, [cond])
        end if
    end for

    return CondDict
end function
```

FIGURE 5.9: Extracting the fixed point condition from a synchroniser (assumes that channel $x$ is declared as an input and an output channel of the synchroniser)

The fixed point condition of an AstraKahn network is formed by the fixed point conditions of its synchronisers that are traversed by the fixed point path. Networks in AstraKahn are represented as graphs, thus the fixed point detection is a graph search problem.

A network graph is a directed multigraph because in AstraKahn two nodes are not restricted to be connected with only one edge. The graph has four types of nodes, namely a box, a synchroniser, a merger and a network. The fixed point path may traverse nodes of any type except for boxes. If the path traverses a node that is a network, the network must have a forward fixed point as well.

The fixed point detection algorithm (Fig. 5.10) is based on the depth-first search algorithm with the following considerations:

- The first and the last nodes of the fixed point path for a particular input channel are known; consequently, only the paths between these two nodes in the graph are traversed
- If the search encounters a box, the traversed path is rejected
- If the search encounters a synchroniser, the fixed point condition of the synchroniser is extracted using the function *extract_fp* in Fig. 5.9. If the synchroniser has no fixed point condition, the traversed path is rejected. Otherwise, the search continues only for the successors of the node that were detected by *extract_fp*
- If the search encounters a merger, it immediately continues to its only successor
- If the search encounters a node that encapsulates a network, the fixed point detection is run on the network. If no fixed point path exists for the network, the traversed path is rejected.

The fixed point detection algorithm runs only on acyclic networks. The wrap-around wiring makes Astra*Kahn* networks cyclic, however, the wrap-around channels cannot carry a fixed point. Therefore, these channels must be filtered before the fixed point detection is run.

### 5.2.2 Discussion

The approach we have presented relies on the ability of synchronisers to encode some checks of the message content and perform different actions depending on the result of a check. As the analysis in previous sections shows, the construction of an operand network with a complex fixed point condition can be quite complicated. In order to avoid having to construct complicated operand networks, we provide an additional fixed point detection strategy that relies on a special port wiring primitive $P$ that transmits messages immediately from one port to another without storing them. The programmer now has to make sure that the fixed point messages are detected within the operand network[6] and sent to $P$. The messages cascade through all the active replicas via a chain

[6] It can be done in a box

**Require:** an operand network graph (*graph*), a channel to test for a forward foxed point ($x$), the first and the last node in the fixed point path (*start*, *end*), the list of fixed point conditions gathered along the path (*cond_list*, optional with the default value *empty_list*)
**Ensure:** a set of fixed point conditions on the channel

```
function DETECT_FFP(graph, x, start, end, cond_list = empty_list)
    if start is a box then
        return empty_list
    end if

    if start is a synchroniser then
        CondDict ← extract_fp(start, x)
        if CondDict = nil then
            return empty_list
        end if
        Lists ← empty_list
        cond_list ← cond_list.Append(start_cond)
        for each succ_node in succ_nodes(start) do
            out_port ← get_label(edge(start, succ_node))
            if out_port ∈ keys(CondDict) then
                if start = end then
                    return cond_list
                end if
                NewLists ← detect_ffp(graph, out_port, succ_node, end, cond_list)
                for new_list in NewLists do
                    Lists ← Lists.Append(new_list) ▷ the fixed point path branches if |Lists| > 1
                end for
            end if
        end for
        return Lists
    end if

    if start is a merger then
        if start = end then
            return cond_list
        end if
        out_port ← get_out_port(start)          ▷ a merger has a single output port
        succ_node ← get_succ_node(start)
        return detect_ffp(graph, out_port, succ_node, end, cond_list)
    end if

    if start is a network then
        start ←get a node that has an input port x
        Lists ← detect_ffp(graph, x, start, end)
        if start = end then
            return cond_list.Append(Lists)
        end if
        return Lists
    end if
end function
```

FIGURE 5.10: A forward fixed point detection in channel $x$ (assumes the network graph is connected)

of $P$ wires and leave the replication network when they encounter an inactive replica. The example in Fig. 5.11 demonstrates how the approach works.

The operand network $A$ in Fig. 5.11 has a single input port $x$ and two output ports. The output port $x$ is intended for the messages that proceed to the next replica of $A$ in the chain, and the output port $x'$ is a auxiliary port for the messages that are supposed to leave the replication pipeline. The serial replication network $A^*$ has a single input and a single output port both named $x$. During the compilation, the operand network $A$ is encapsulated into the special network $N$ it as shown in Fig. 5.11. The network $N$ inherits all the ports from $A$ and adds the corresponding input port $x'$. The input and output ports $x'$ of $N$ are connected with the wiring primitive $P$. The output and the input ports $x'$ of the consequent replicas of $N$ are connected with the wiring primitive $P$ as well. A message that $A$ sends to the output port $x'$ cascades through all the

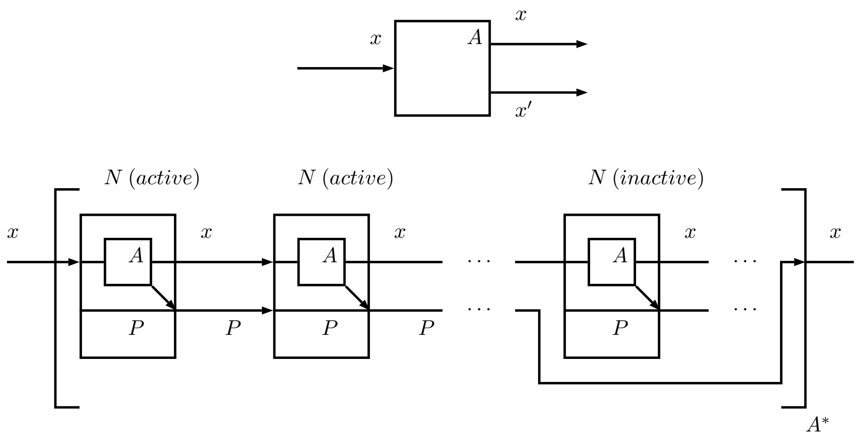


FIGURE 5.11: The operand network $A$ (top) and a possible implementation of its serial replication $A^*$ (bottom)

active replicas. Once it has reached an inactive replica, the output channel $x$ of $A^*$ is dynamically wired to the output port $x'$ of the last active replica of $N$. When an inactive replica becomes active, the port $x'$ is rewired with the input port of this replica using $P$.

## 5.3 Reverse Fixed Point

A reverse fixed point on channel $x$ is a state of a replica, in which it transmits messages from channel $x$ unchanged. A state of a replica is formed by the states of its synchronisers. Astra*Kahn* does not analyse boxes and it can determine the operand network behaviour only from its synchronisers. Thus, in order for the operand network to be analysable by Astra*Kahn* it must contain a path that does not traverse boxes and which may traverse synchronisers. Every synchroniser that is traversed by the path must have at least one state, in which it accepts a message from channel $x$ unconditionally and sends it on to the next synchroniser along the path without storing or modifying the message.

The existence of a reverse fixed point state requires the operand network to have some topological properties that are formally defined as follows. Consider a network $N$ that has an input and an output channel, both named $x$.

**Definition 2.** The network $N$ is said to have a reverse fixed point in $x$ if and only if the following requirements are satisfied:

1. A unique non-branching path from the input to the output channel $x$ exists that does not traverse any boxes
2. Every synchroniser $S_i$ on the path has a subset of states, which we denote as $s_i$, such that in each of these states every message on the path is immediately transferred without being changed or stored, causing the synchroniser to remain in the same state[7]. In a state from $s_i$ the synchroniser $S_i$ may still be sensitive to other input channels, as long as this does not, under any circumstances, cause a transition to a state outside $s_i$

The network $N$ is said to be in a reverse fixed point state on channel $x$ when each $S_i$ is in a state that belongs to its $s_i$.

### Rewiring of the Reverse Fixed Point

The reverse fixed point optimises an input connection that has to cascade through the chain to a replica that is ready to accept the data.

The operand network $N$ in Fig. 5.12 has two input and two output channels $x$ and $y$. Any input channel $x$ wired to an active replica of $N$ that transitions to a reverse fixed point state on that channel is disconnected from the replica and dynamically rewired to the input port $x$ of the next replica in the chain.

If a replica transitions to a reverse fixed point state on every one of its input channels, no box is running and the channels are empty, the replica is removed from the chain.

[7] We shall note that the values of any state variables form a part of the synchroniser state

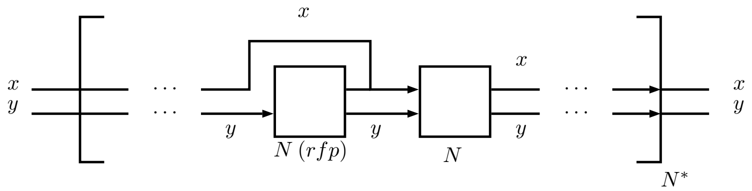


FIGURE 5.12: Rewiring of a Reverse Fixed Point replica $N\ (rfp)$ on channel $x$

### 5.3.1 Reverse Fixed Point Detection

The existence of a reverse fixed point on channel $x$ requires the synchronisers that are traversed by the fixed point path to have at least one state, in which they unconditionally accept messages from that channel, send them on along the path unchanged and then transition back to the same state. A synchroniser that is in the reverse fixed point state never transitions from it. Coded in the synchroniser language, a transition that makes a state of a synchroniser the reverse fixed point state on channel $x$ is given in Fig. 5.13.

```
s {
  on:
    x {
      send this => out;
      goto s;
    }
  ...
}
```

FIGURE 5.13: The reverse fixed point state $s$ of a synchroniser on channel $x$

Note that all other transitions from state $s$ must come back to state $s$. As long as they do they can even change the state variables of the synchroniser; the reverse fixed point is unconditional and it exists for any values of the store variables.

The algorithm in Fig. 5.14 checks if a reverse fixed point exists on channel $x$ and extracts all fixed point states from the synchroniser source code. The algorithm is designed to work with the detection algorithm in Fig. 5.10.

**Require:** the abstract syntax tree of the synchroniser program (*synch*), the input label of a channel to test for a forward fixed point ($x$)
**Ensure:** a dictionary ($a, StateList$), where $a$ is the output label of the fixed point channel and $StateList$ is the list of the reverse fixed point states of the synchroniser

```
function EXTRACT_FP(synch, x)
    rfp_states ← all states from synch that have transitions like in Fig. 5.13
    result ← ∅

    while rfp_states ≠ result do
        if result ≠ ∅ then
            rfp_states ← result
            result ← ∅
        end if
        for each state in rfp_states do
            gotos ← all destination states in state
            if gotos \ rfp_states = ∅ then
                result ← result ∪ state
            end if
        end for
        if result = ∅ then
            return no reverse fixed point state found
        end if
    end while

    StateDict ← nil
    for each state in result do
        rfp_out ← output channel labels of the RPF transitions from state
        for each out in rfp_out do
            if out ∉ StateDict then
                StateDict(out)← state
            else
                StateDict(out)← StateDict(out).Appendstate
            end if
        end for
    end for

    return StateDict
end function
```

FIGURE 5.14: Extracting the reverse fixed point from a synchroniser (assumes that channel $x$ is declared as an input and an output channel of the synchroniser)

# Chapter 6

# Conclusion

## 6.1 Summary

The implementation of Astra*Kahn* synchronisers and analysis of their role in the serial replication wiring pattern have been presented. A dedicated language that Astra*Kahn* provides for programing synchronisers was described in details. A synchroniser exploits non-deterministic behaviour and in order to explain how the synchroniser makes choices, the synchronisers execution algorithm has been given. The language compiler that was implemented in the thesis project generates the data structure to be interpreted by the Astra*Kahn* runtime, and the communication passport of the synchroniser. The compiler performs static checking and reports source code errors. The compiler can be used in checking of the static correctness of a connection between components all over the application network.

In Astra*Kahn* the output from the serial replication pipeline is defined using the concept of fixed point. In order to detect fixed point messages, Astra*Kahn* needs to be provided with a pattern that matches all of them. This thesis has shown exactly how this pattern can be embedded into the operand network of the serial replication combinator, so that the programmer does not have to specify it explicitly within the Astra*Kahn* application code. However, the analysis has shown that the original approach to the output from the serial replication network can be quite inconvenient for code maintenance and debugging when the fixed point condition is complex, so the thesis has suggested a simpler fixed point detection strategy in addition to the original one. In order to suppress the growth of the replica chain, Astra*Kahn* introduces a reverse fixed point concept. The thesis has presented an example Astra*Kahn* code to motivate the reverse fixed point and has shown how the reverse fixed point can be used. As the result of the analysis performed in the thesis, the forward and the reverse fixed point detection algorithms to be implemented in the Astra*Kahn* compiler have been provided.

## 6.2 Future Work

The current version of the synchroniser language does not define flow inheritance in synchronisers, thus the next step in the synchroniser implementation is to decide how it should be done. The synchroniser code only needs to access the label-value pairs of the message it matches. Thus, the flow inheritance should be supported outside the synchroniser definition.

The synchroniser that was presented matches only label-value pairs of a record. The MDL, which is the basis of the Astra*Kahn* type system, generates a much broader set of terms than the current version synchroniser can process; however, whether or not the synchronisation in record values is useful for the real world applications still needs to be established.

The fixed point detection algorithms given in the thesis should be implemented in the Astra*Kahn* compiler. The dynamic rewiring support for the serial replication pattern should be implemented in the Astra*Kahn* runtime system.

The long-term goal of the Astra*Kahn* project is to provide an environment for development of scalable concurrent applications that do not require manual tuning. One approach to address automatic concurrency management is statistical learning. In a streaming network the time during which a particular box or synchroniser processes messages is distributed according to some distribution. On the other hand, processes translate the input distribution to the output distribution to some degree depending on their definition. Knowing these characteristics for both boxes and synchronisers, it could be possible to estimate the parallelisation factors for each box in order to optimise the throughput or the latency of the network.

# Appendix A

# Syntax of the **Astra***Kahn* Syncroniser

## A.1 Full Grammar

The full grammar of the synchroniser language can be found in Fig. A.1

The grammar of integer expression used in the synchroniser implementation can be found in Fig. A.2.

## A.2 Keywords, Reserved Words and Punctuation

The keywords, the reserved words and the punctuation used in the Astra*Kahn* synchroniser syntax are given in Fig. A.3.

## A.3 Abstract Syntax Tree of the Synchroniser

The *aksync* compiler uses the abstract syntax tree (AST) code representation. The AST nodes represent significant programming constructs of the synchroniser language. In our implementation we use the AST generator by E. Bendersky [18]. The generator configuration file (Fig. A.4) describes the structure of the AST nodes. Each node lists the attributes and the child nodes. A single star (*) and two stars (**) depict a single child node and a sequence of nodes of the same type respectively. A string depicts an attribute.

```
⟨sync⟩          ::= ‘synch’ ⟨ID⟩ ‘(’ ⟨input⟩ [‘,’ ⟨input⟩]* ‘|’ ⟨output⟩ [‘,’ ⟨output⟩]* ‘)’
                    ‘{’ ⟨decl⟩* ⟨state⟩+ ‘}’

⟨input⟩         ::= ⟨ID⟩ [‘:’ (⟨ID⟩ | ⟨NUMBER⟩)]

⟨output⟩        ::= ⟨ID⟩ [‘:’ ⟨depth_exp⟩]

⟨depth_exp⟩     ::= ⟨ID⟩ | ⟨NUMBER⟩ | ⟨ID⟩ ‘+’ ⟨NUMBER⟩ | ⟨ID⟩ ‘-’ ⟨NUMBER⟩

⟨decl⟩          ::= ‘store’ ⟨id_list⟩ ‘;’
                  | ‘state’ ⟨type⟩ ⟨id_list⟩ ‘;’

⟨type⟩          ::= ‘int’ ‘(’ ⟨NUMBER⟩ ‘)’
                  | ‘enum’ ‘(’ ⟨id_list⟩ ‘)’

⟨state⟩         ::= ⟨ID⟩ ‘{’ ‘on:’ ⟨trans_stmt⟩+ [‘elseon:’ ⟨trans_stmt⟩+]* ‘}’

⟨trans_stmt⟩    ::= ⟨ID⟩ [‘.’ ⟨condition⟩] [‘&’ ⟨int_exp⟩ ] ⟨actions⟩

⟨condition⟩     ::= ‘@’ ⟨ID⟩
                  | ‘?’ ⟨ID⟩
                  | [‘?’ ⟨ID⟩] ‘(’ ⟨id_list⟩ [‘||’ ⟨ID⟩ ]‘)’
                  | ‘else’

⟨actions⟩       ::= ‘{’ [⟨set_stmt⟩] [⟨send_stmt⟩] [⟨goto_stmt⟩] ‘}’

⟨set_stmt⟩      ::= ‘set’ ⟨assign⟩ [‘,’ ⟨assign⟩]* ‘;’

⟨assign⟩        ::= ⟨ID⟩ ‘=’ (⟨int_exp⟩ | ⟨data_exp⟩)

⟨send_stmt⟩     ::= ‘send’ ⟨dispatch⟩ [‘,’ ⟨dispatch⟩]* ‘;’

⟨dispatch⟩      ::= ⟨msg_exp⟩ ‘=>’ ⟨ID⟩

⟨msg_exp⟩       ::= ‘@’ ⟨ID⟩
                  | ‘@’ ⟨int_exp⟩
                  | [‘?’ ⟨ID⟩] ⟨data_exp⟩
                  | ‘nil’

⟨data_exp⟩      ::= ⟨data⟩
                  | ‘(’ ⟨data⟩ ‘)’

⟨data⟩          ::= ⟨item⟩ [‘||’ ⟨item⟩]*

⟨item⟩          ::= ‘this’
                  | ⟨ID⟩
                  | ‘'’ ⟨ID⟩
                  | ⟨ID⟩ ‘:’ ⟨rhs⟩

⟨rhs⟩           ::= ⟨int_exp⟩
                  | ⟨ID⟩

⟨goto_stmt⟩     ::= ‘goto’ ⟨id_list⟩ ‘;’

⟨id_list⟩       ::= ⟨ID⟩ [‘,’ ⟨ID⟩]*

⟨int_exp⟩       ::= ‘[’ ⟨int_exp_c⟩ ‘]’
```

Figure A.1: The syntax of the AstraKahn synchroniser

```
⟨int_exp_c⟩   ::= ⟨NUMBER⟩
                |  ⟨ID⟩
                |  '(' ⟨int_exp_c⟩ ')'
                |  ⟨int_exp_c⟩ '+' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '-' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '*' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '/' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '%' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '<<' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '>>' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '|' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '&' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '^' ⟨int_exp_c⟩
                |  '-' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '<' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '>' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '==' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '!=' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '<=' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '>=' ⟨int_exp_c⟩
                |  '!' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '&&' ⟨int_exp_c⟩
                |  ⟨int_exp_c⟩ '||' ⟨int_exp_c⟩
```

FIGURE A.2: The syntax of the integer expression in Astra*Kahn* synchroniser

| | |
|---|---|
| Keywords | synch, store, state, int, enum, start, on, elseon, else, do, send, goto |
| Reserved words | nil, this |
| Punctuation | braces, brackets, parantheses, the comma, the dot, the semicolon, the plus sign, the minus sign, the ampersand, the at sign, the question mark, the bar-bar sign, the equality sign, the arrow |

FIGURE A.3: Astra*Kahn* synchroniser keywords, reserved words and punctiation

```
# inputs -> PortList, outputs -> PortList, decls -> DeclList, states -> StateList
Sync: [name, inputs*, outputs*, decls*, states*]

# ports -> [Port, ...]
PortList: [ports**]

# depth_exp -> ID | NUMBER | DepthExp | DepthNone
Port: [name, depth_exp*]
DepthExp: [depth, shift]
DepthNone: []

# decls -> [StoreVar | StateVar, ...]
DeclList: [decls**]

StoreVar: [name]

# type -> IntType | EnumType
StateVar: [name, type*]
IntType: [size]

# labels -> [ID, ...]
EnumType: [labels**]

# states -> [State, ...]
StateList: [states**]

# trans_orders -> [TransOrder, ...]
State: [name, trans_orders**]

# trans_stmt -> [Trans, ...]
TransOrder: [trans_stmt**]

# condition -> CondSegmark | CondDataMsg | CondEmpty | CondElse
# guard -> IntExp
# actions -> [Assign | Send | Goto, ...]
Trans: [port, condition*, guard*, actions**]

CondSegmark: [depth]

# labels -> [ID, ...]
CondDataMsg: [choice, labels**, tail]

CondEmpty: []
CondElse: []

# rhs -> DataExp | IntExp
Assign: [lhs, rhs*]

# items -> [ItemThis | ItemVar | ItemExpand | ItemPair, ...]
DataExp: [items**]
ItemThis: []
ItemVar: [name]
ItemExpand: [name]
# value -> ID | IntExp
ItemPair: [label, value*]

# msg -> MsgSegmark | MsgData | MsgNil
Send: [msg*, port]

# depth -> ID | IntExp
MsgSegmark: [depth*]

# data_exp -> DataExp
MsgData: [choice, data_exp*]
MsgNil: []

# states -> [ID, ...]
Goto: [states**]

ID: [name]
NUMBER: [value]
IntExp: [exp]
```

FIGURE A.4: The generator configuration file of the synchroniser AST nodes. Child node types are provided in comments (marked with dashes).

# Bibliography


[1] Alex Shafarenko. Astrakahn: A coordination language for streaming networks. *CoRR*, abs/1306.6029, 2013.

[2] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *ADVANCES IN COMPUTERS*, pages 329–400. Academic Press, 1998.

[3] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, August 1986.

[4] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[5] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[6] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, July 2006.

[7] Robert Stephens. A survey of stream processing. *Acta Inf.*, 34(7):491–541, 1997.

[8] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, 1977.

[9] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[10] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, January 1987.

[11] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[12] Clemens Grelck, Sven-Bodo Scholz, and Alexander V. Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[13] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Concurrency Engineering with S-Net. In Adrian Prantl Jens Knoop, editor, *Programming Languages and Foundations of Programming, 15th Workshop (KPS'09), Maria Taferl, Austria*, volume 2009-X-1 of *Technical Report*, pages 78–92. Institute of Computer Languages, Vienna University of Technology, 2009.

[14] Clemens Grelck. The Essence of Synchronisation in Asynchronous Data Flow Programming. In Jurriaan Hage, editor, *22nd International Symposium on Implementation and Application of Functional Languages (IFL'10), Alphen aan den Rijn, Netherlands*, volume UU-CS-2010-020 of *Technical Report*, pages 159–172. Department of Information and Computing Sciences, Utrecht University, 2010.

[15] Vu Thien Nga Nguyen and Raimund Kirner. Demand-based scheduling priorities for performance optimisation of stream programs on parallel platforms. In *ICA3PP (1)*, pages 357–369, 2013.

[16] David Beazly et al. Ply (python lex-yacc). http://www.dabeaz.com/ply/, February 2011. [Online; accessed 9-December-2014].

[17] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[18] Eli Bendersky. A parser for the c language written in pure python. https://github.com/eliben/pycparser, 2015.