A Survey Of Stream Processing *

R. Stephens
Computer Systems Research Group,
Department of Electronic and Electrical Engineering,
University of Surrey,
Guildford,
Surrey.
GU2 5XH.

email: r.stephens at ee.surrey.ac.uk

October 24, 1995

Abstract

Stream processing is a term that is used widely in the literature to describe a variety of systems. We present an overview of the historical development of stream processing and a detailed discussion of the different languages and techniques for programming with streams that can be found in the literature. This includes an analysis of *dataflow*, *specialized functional and logic programming with streams*, *reactive systems*, *signal processing systems*, and the use of streams in the design and verification of hardware.

The aim of this survey is an analysis of the development of each of these specialized topics to determine if a general theory of stream processing has emerged. As such, we discuss and classify the different classes of stream processing systems found in the literature from the perspective of programming primitives, implementation techniques, and computability issues, including a comparison of the semantic models that are used to formalize stream based computation.

*To appear in *Acta Informatica*. This report is a revised version of Reports CSRG95-03 and CSRG95-04.

Contents

1 Introduction 1
1.1 A Theory of Stream Processing . . . 2
1.2 Overview . . . 2
1.3 Acknowledgements . . . 3

2 A Brief History of Stream Processing 3
2.1 The 1960s . . . 3
2.2 The 1970s . . . 3
2.3 The 1980s . . . 4
2.4 The 1990s . . . 5

3 Dataflow 5
3.1 Origins . . . 6
3.2 Dataflow Networks . . . 6
3.3 Dataflow Computation and Semantics . . . 6
3.4 The Uptake of Dataflow . . . 7
3.5 Synchronous Dataflow . . . 8

4 Specialized Functional and Logic Programming 8
4.1 Overview . . . 8
4.2 Functional Approaches to Stream Processing . . . 8
4.3 Logic Programming Languages with Streams . . . 10

5 Reactive Systems and Signal Processing Networks 11
5.1 Streams, Signals and Sensors . . . 11
5.2 The Strong Synchrony Hypothesis and Multiform Time . . . 11

6 Stream Processing in the Design and Verification of Hardware 12
6.1 SCAs . . . 12

7 Other Stream Processing Formalisms 13
7.1 ALPHA . . . 13
7.2 Stream X-Machines . . . 14

8 Stream Processing Primitives and Constructs 14
8.1 Introduction . . . 14
8.2 Common Functional Stream Processing Operations . . . 14
8.3 Stream Processing Primitives in Logic Programming . . . 16

9 Stream Processing Languages 19
9.1 A Running Example: the RS-Flip-Flop . . . 19
9.2 Formalization of the Flip-Flop as a ST . . . 19
9.3 An Implementation of the Flip-Flop as a SPS . . . 20
9.4 Lucid . . . 21
9.5 LUSTRE . . . 23
9.6 Other Dataflow Languages . . . 26

9.7 SIGNAL . . . 26
9.8 ESTEREL . . . 29
9.9 AL . . . 32
9.10 PL . . . 33
9.11 Daisy . . . 35
9.12 PROLOG with streams . . . 35
9.13 STREAM . . . 37
9.14 ASTRAL . . . 40

10 Conclusions **42**

References **44**

1 Introduction

Within computer science the term *stream processing* is used generically to refer to the study of a number of disparate systems. For example, *dataflow systems*, *reactive systems*, *synchronous concurrent algorithms*, *signal processing systems*, and certain classes of *real-time systems* are all examples of stream processing research.

At the conceptual level the mathematical analysis of each of these systems is usually based on the study of a particular type of *stream processing system* (SPS); that is, it is based on the study of a system comprised of a collection of *modules* that compute in parallel, and that communicate data via *channels*. In particular, in a typical SPS modules are usually divided into three classes: *sources* that pass data into the systems; *filters* (also called *agents*) that perform some atomic computation; and *sinks* that pass data from the system. SPSs are often visualized as directed graphs, for example, a three-source, two-sink SPSs with five filters is shown in Figure 1.

SPSs take their name from the communication performed by their channels that pass information between modules as infinite sequences of data that are referred to as *streams*. A *stream*, is essentially an infinite list of elements $a_0, a_1, a_2, \ldots$ taken from some data set of interest A, and is usually formalized mathematically as a function $a : T \rightarrow A$, wherein $T = \mathbb{N} = \{0, 1, 2, \ldots\}$ represents discrete time.

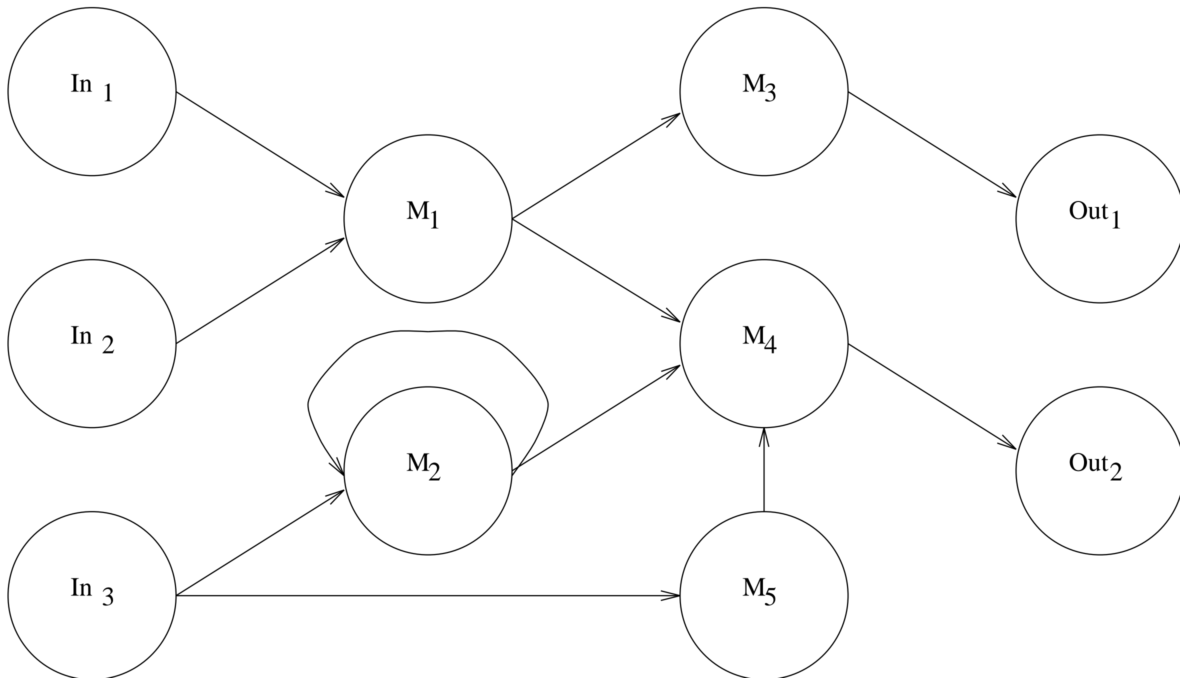

Figure 1: A Typical SPSs

Stream processing research, in particular the study of SPSs, can be traced back at least as far as the 1960s, although not always in a form that is immediately recognizable as such today. Indeed, stream processing has been a particularly active area of research as the visualization of systems as SPSs is appropriate to formalize many types of computational models that arise quite naturally in computer science including: *artificial neural networks*, *coupled map lattice*

dynamical systems, *cellular automata* and also operating systems and many types of *safety critical systems*. Moreover, stream based computation is also appropriate to formalize hardware at several levels of abstraction including the *conceptual level* and the *register transfer level*.

1.1 A Theory of Stream Processing

Despite the usefulness of stream based computation as a conceptual tool, in our opinion the preoccupation of stream processing with SPSs has from some perspectives hindered the development of a clear, concise and mathematically neutral theory of systems that compute over streams. We believe this observation is supported by the large variety of different implementation techniques and semantic models that can be found in the literature to model SPSs, whose advantages over other techniques are often justified using qualitative rather than quantitative arguments. Indeed, we believe it is fair to observe that with a few exceptions stream processing systems have been used as a convenient tool in research that has been concerned with other issues, rather than the development of a *general theory* of stream processing encompassing topics such as: the analysis of the computability of stream processing primitives and stream based computation; a study of the languages and logics needed to specify and reason about systems that compute over streams; and the theory of the verification of different classes of stream processing systems.

For example, dataflow is considered to be a canonical example of stream processing research, but dataflow is predominately concerned with the development of parallel processing techniques. In particular, this is highlighted by the fact that dataflow is often considered to be a specialized implementation method for functional programming rather than a separate area of research.

As another example, M Broy has carefully developed an extensive theory of parallel, distributed, asynchronous SPSs using functional techniques (see Section 4.2.2). However, this work is by definition also specialized and hence does not provide a general theory of stream processing in the sense defined above (see [45]).

1.2 Overview

To clarify some of the issues that we believe are important in the development of a general theory of stream processing, in this paper we present a survey of the literature. It is our aim to highlight some important theoretical as well as practical considerations and so our discussions are based on the analysis of *stream transformers* (STs) of which SPSs can be considered as a special case. Specifically, a ST is an abstract system that takes n streams as input and produces m streams as output for some $n, m \geq 1$, and can be characterized as a *functional*

$$\Phi : [T \to A]^n \to [T \to A]^m.$$

In contrast, a SPS is a system composed of a collection of separate, but communicating processes that receive stream data as input and produce stream data as output. Thus, a SPS can be viewed as a (parallel) *implementation* of an abstract ST *specification*, and *stream processing* can be defined as the study of both STs and SPSs.

1.2.1 Summary.

In Section 2 we have a brief historical perspective of the development of stream processing from the early 1960s to the present day.

The three sections following this historical overview are devoted to a more detailed analysis of some of the different approaches to visualizing and representing SPSs: *dataflow* (Section 3); specialized functional and logic programming (Section 4); reactive systems and signal processing

(Section 5); and stream processing in the design of hardware (Section 6). In each case we discuss the basic motivations and ideas underlying each paradigm. However, for convenience in order to clarify certain issues relating to computability theory and language design we have deferred the topic of stream processing primitives and languages arising in this research until Sections 8 and 9. Our literature survey is concluded in Section 7 wherein we briefly mention some topics related to stream processing.

In Section 10 we make some concluding remarks.

1.3 Acknowledgements

I would like to thank the following colleagues for their comments and suggestions during the preparation of this survey: B C Thompson, B R McConnell, M J Poole (Swansea) and L J Steggles (Newcastle). In addition, I would also like to thank the (anonymous) referees for their comments that have improved the presentation of this paper. In particular, I thank S D Johnson (Indiana) for his very detailed and constructive comments.

2 A Brief History of Stream Processing

In this section we present a brief historical perspective of the development of stream processing over the last four decades. We note that we only mention either well-known research or research that we believe is representative of a particular topic within stream processing, and that provides a useful starting point for any further reading the reader may wish to undertake. Some of the topics covered in our overview are analysed in more depth in the following sections.

2.1 The 1960s

Within computer science the term *stream* has been attributed to P J Landin (see [50]) formulated during the development of operational constructs presented as part of his work on the correspondence between ALGOL 60 and the λ-calculus (see [138] and [139]). Indeed, we note that P J Landin's original use for streams was to model the histories of loop variables, but he also observed that streams could have been used as a model for I/O in ALGOL 60.

The first type of SPSs that can be identified within the literature are *dataflow systems* that have certainly existed, although not always under the name 'dataflow', as early as the late 1960s (see for example [157] and [5]). The term dataflow originates from the term *data flow analysis* (see [3]) used to evaluate potential concurrency in computations.

2.2 The 1970s

The first dataflow language, and probably still the most famous, is Lucid (see [223]) that was conceived in 1974. Lucid is based in part on the language POP-2 (see [51]), that allowed a limited use of streams. Other relevant dataflow references from the 1970s are [6], [135], [68], [226] and [10]).

In 1974 G Kahn published his well-known work (see [121]) outlining a simple parallel programming language designed for representing SPSs using a fixed-point semantics. The use of a fixed-point semantics for SPSs in the style of Kahn's work is common, and for this reason SPSs are sometimes referred to as Kahn networks.

In 1975 W Burge (see [50]) discussed the use of streams as a method for structured programming and introduced a set of functional stream primitives for this purpose.

In 1976 P Henderson and J H Morris (see [100]) and D P Friedman and D S Wise (see [78]) published their work on lazy evaluation techniques that are useful for computing with infinite data types of which streams are an example. The work presented in [78] was the first in the research project 'applicative programming for streams' examining the issue of concurrency using stream programming. This work is continued in [79] that presents a file system and text editor, and [79] that explores the use of streams in programming problems such as approximations to real arithmetic, the Sieve of Eratosthenes and 2-3-5 composites.

In 1977 G Kahn made another contribution to the field with his joint paper with D MacQueen (see [123]) wherein they introduced a language designed to model distributed process interaction using ideas from [121].

2.3 The 1980s

Dataflow continued to be an area of widespread research during the 1980s and several additional semantic models for dataflow were introduced, for example, [76], [21], [197], [198], [199], [131] and [120].

Logic programming languages also began to be used to model SPSs. A modification of PROLOG used to model what have become termed *perpetual processes* (see [147]) within logic programming was introduced in [19].

Functional programming languages were also used widely to model SPSs. Notable in this area is the work of M Broy and his use of functional languages to study stream based distributed processing (see for example [35], [36], [37], and [38]).

In 1985 the first paper on the subject of *synchronous concurrent algorithms* (SCAs) was released. Conceived by B C Thompson and J V Tucker, SCAs (see [209]) have been the stimulation for much of our own work into stream processing.

The year 1985 also saw the publication of a paper by D Harel and A Pneuli (see [91]) on the subject of *reactive systems*. Reactive systems, together with *signal processing networks* and *synchronous dataflow networks* – that can be considered as special cases of reactive systems – have been the stimulation for a large body of stream processing research (see for example [84], [53] and [25]).

During the 1980s streams and STs have also been used extensively for hardware description, for example, [192], [193], [194], [65], [66], [95], [177] and [92]. We note that the work of M Sheeran on Ruby, as discussed in [192], [193] and [194] above, includes a generalization of the representation of streams to the function space $[\mathbb{Z} \rightarrow A]$ (wherein $\mathbb{Z}$ represents the integers) to avoid dealing with initial conditions in hardware specifications. (References on the incorporation of streams into foundational mathematics can be found in Section 10.)

Two surveys of the use of stream processing in hardware design during the 1980s can be found in [176] and [64]. Indeed, the work presented in [176] is based in part on the work of D P Friedman and D S Wise and is the first in a large body of research concerned with the use of applicative stream processing for the design and synthesis of hardware based on the language *Daisy*.

2.4 The 1990s

As with the 1980s, semantic models for dataflow are still being developed, for example, [125], [14], [15], [77], and [141], although the work in [14], [15] is concerned with *flowchart schemes* (see [142]) that has applications to the study of dataflow schemes. A useful overview of the concept of dataflow with an extensive bibliography can be found in [190].

Developments of Ruby have continued including formalizations of the Ruby algebra (see for example [195] and [191].)

SCAs continue to be an intensive area of research (see Section 6.1 for references) as does research into reactive systems (see for example [28], [88], [85], [184] and [87]).

The 1990s have also produced a body of work concerned with the theoretical foundations of stream processing. In 1992 J V Tucker and J I Zucker (see [213]) released the first in a series of generalizations of computability theoretic results from the natural numbers to algebras with streams. This work is continued in [214]. In addition, [201] presents a theoretical study of the compositional properties of STs in *Cartesian form*.

The theoretical work of K Meinke has applications to the specification, verification and parameterization of STs (see [160], [161], [162], [163]) as does the work of K Meinke with L J Steggles and B M Hearn (see [164] and [99] respectively).

Finally, M Broy continues his functional study of distributed processing over streams (see for example [42], [46], and in particular [45]).

3 Dataflow

As dataflow networks were the first type of SPSs to appear in the literature we begin our more detailed survey with an examination of the research aims of dataflow and an analysis of the semantic models and implementation techniques that have been developed. A more detailed introduction to the concept of dataflow can be found in [190].

To aid in this discussion, because so much of the stream processing research in the literature is concerned with SPSs, in the sequel it is useful to be able to identify different types of SPSs concisely. Therefore, we will classify SPSs by the following three main characteristics:

(1) Either *synchronous* or *asynchronous* filters – that is, filters that either compute in a synchronized manner with respect to other filters (as for example in Section 6.1) or filters that compute with no synchronization with respect to other filters.

(2) Either *deterministic* or *non-deterministic* filters – that is, filters that either do or do not compute a function.

(3) Either *uni-directional* or *bi-directional* channels.

Notice that in our definitions each of these three classifying features only reflect the behaviour of individual filters and not the whole SPS of which they are a part. Thus, from the perspective of our classification synchrony and asynchrony are independent from determinism and non-determinism, whereas from the perspective of the whole network behaviour this may not be the case.

We will use the following shorthand notation to denote SPSs that are designed to model networks with specific combinations of the above three properties: $\varsigma\delta\upsilon$-SPS, wherein $\varsigma \in \{S, A\}$, $\delta \in \{D, N\}$, and $\upsilon \in \{U, B\}$. For example, using this classification a synchronous, deterministic SPS with unidirectional channels is denoted SDU-SPS, and an asynchronous, non-deterministic SPS with bidirectional channels is denoted ANB-SPS.

3.1 Origins

As we have mentioned dataflow research began as far back as the 1960s and continues to be an area of widespread research. One of the continuing aims of the dataflow approach has been to avoid the so-called 'von Neumann bottleneck' (see [12] and [13]) and exploit the parallelism offered by VLSI technology. As part of this research many experiments with specialized architectures have been undertaken (the interested reader can consult the bibliography of [190] for a list of references).

We note in passing at this point that it has been observed that the link between J von Neumann and sequential computing methods is historically inaccurate, as he was one of the early advocates of parallel computing methodologies ([127]). However, we use this phrase as it can be found in the dataflow literature.

3.2 Dataflow Networks

A classical dataflow network is an ADU-SPS, although dataflow computation based on ANU-SPSs has also been studied, and more recently dataflow computation based on SDU-SPSs has been of interest. The filters within a dataflow network (sometimes referred to as *coroutines* – see [157]– and also *agents*) compute over streams – that is, A^ω (see below) wherein the data type A is usually restricted to *int*, *bool*, *real* and lists of these types.

3.3 Dataflow Computation and Semantics

From an operational perspective the dataflow model of computation is typically divided into two basic forms: *data driven* (eager evaluation) wherein filters compute depending upon the availability of data at their inputs; and *demand driven* (lazy evaluation) wherein filters request data on the input lines when they wish to compute. However, in [112] it is argued that neither of these informal implementation ideas embody a semantics that is suitable for dataflow computers. In contrast, [112] suggests four operational semantic models for dataflow: *piped eager*, *tagged eager*, *piped lazy* and *tagged lazy* based on a graphical language called *flat operator nets*. Furthermore, [112] shows that using such a model it is possible to determine whether the operational semantics of a dataflow network is equivalent to its intended denotational semantics.

3.3.1 Kahn's Work. The most common approach to a denotational model for dataflow is a domain-theoretic semantics based on the influential work of G Kahn (see [121]), wherein he introduced a simple parallel language for representing ADU-SPSs in an ALGOL like style. However, the generality of the method Kahn introduced to provide a semantics for this language means that his techniques can also provide a language independent semantic model for both SDU-SPSs and some ADU-SPSs (see Section 3.3).

Kahn's interest in such a language was not motivated by the development of a user-friendly programming methodology for describing ADU-SPSs, rather Kahn was interested in how to

prove formally properties of programmes written in such a language. In particular, properties relating to the networks they described including termination, non-termination and properties of their output.

Kahn observed that the (possible infinite) sequences of data from some data type D passing along arcs (what he referred to as *histories*) that connect modules in a SPS can be formalized mathematically as the set $D^\omega = [T \to D] \bigcup D^*$ (see Section 8.2). As such by associating the usual partial ordering with D^ω he observed that D^ω is a *complete partial order*. Therefore, it is possible to view a SPS N with $n \in \mathbb{N}$ input arcs and $m \in \mathbb{N}$ output arcs as computing a functional

$$F^N : (D^\omega)^n \to (D^\omega)^m.$$

Hence, with the assumption that each module in N computes a continuous function (not an unreasonable assumption) the functional F is itself continuous and so it is possible to apply the *First Recursion Theorem* (see for example [167], [59] and [202]) to derive a semantics for N. More specifically, Kahn observed that the *least fixed point* of the functional F^N is the required semantics of the network N. Moreover, Kahn showed that using this method the step from formalizing the semantics of SPSs to formalizing the semantics of the language he introduced for describing such networks (or indeed any well-typed language for describing SPSs) is straightforward.

As such, in line with Kahn's motivations he observed that by adopting a fixed-point semantic approach that Scott's Induction Rule (Kahn cited [148]) and several techniques for proving properties of recursive programs found in [220] are now available to the programmer, including structural induction and recursion induction.

3.3.2 Other Dataflow Semantical Models.

Despite the generality of Kahn's method it is not appropriate for some more general classes of dataflow network (see [132]). For example, non-deterministic models of dataflow computation. Furthermore, 'straightforward' extensions to the Kahn semantic model to cope with non-determinism can fail to be compositional (see [34] and also [183]). Consequently for this and other reasons many other semantic models have been formulated for dataflow. For example, some recent references include [76], [21], [197], [131], [112], [125], [141], and [77].

Despite the many semantic models for dataflow none seems to have been widely adopted. In addition, while the equivalence of certain operational and denotational semantic models for dataflow is addressed (see for example [76], [112], [126], [36] and [40]) the formal relationship between the many different approaches to dataflow is poorly addressed in the literature, as is the correspondence between the model of computation provided by dataflow and formal models of computation. For example, it is interesting that despite the fact that the dataflow approach is in many senses closely related to CCS (see [170]) that (as far as we are aware) dataflow has not been formalized using this well-developed formalism. It has been suggested that this is due to the 'value' passing nature of dataflow networks that CCS does not handle concisely; and that the modelling of dataflow creates too much overhead to make its mathematical formulation using CCS worthwhile.

3.4 The Uptake of Dataflow

Despite the extensive body of dataflow research the dataflow approach appears to have had little impact on the traditional approach to 'von Neumann computing'. Indeed, even the most

well-known dataflow language Lucid (see Section 9.4) has been described as 'a language looking for an application'.

The reasons for the poor uptake of the dataflow approach may stem from two different areas: on the practical side, implementation of the dataflow approach on conventional architecture leads to inefficiencies, including large and wasteful memory usage, that has required the development of specialized architectures; and on the theoretical side, as we have already mentioned, the lack of a generally accepted clear and straightforward semantics.

3.5 Synchronous Dataflow

The asynchronous nature of dataflow can lead to problems with non-determinism and associated anomalous behaviour (see [42]); and cyclic networks can suffer from deadlock (see [222] and [172]). Synchronous dataflow has been developed to avoid these problems. While each filter in a synchronous dataflow network still has its own clock, rather than a global clock as the name might suggest, the interplay between these clocks is restricted and ensures synchronous (and hence deterministic) behaviour.

We discuss synchronous dataflow more fully in Section 9.5 when we examine the language LUSTRE that is used to describe synchronous dataflow networks.

4 Specialized Functional and Logic Programming

The theoretical approaches used to incorporate streams into functional and logic programming languages are in many cases closely related, and are essentially that of a domain-theoretic approach. For this reason we have grouped these two areas of research together into a separate section of our literature survey. A detailed discussion of the domain-theoretic relationship between functional and logic programming languages can be found in [196].

4.1 Overview

In Section 4.2 we examine the functional approach to stream processing and in particular the work of M Broy.

In Section 4.3 we look and logic programming with streams, and in detail at a modification of PROLOG that can be used for stream processing.

4.2 Functional Approaches to Stream Processing

The use of the function abstraction operator (λ-abstraction) provides a mechanism for the representation of STs in functional languages in both second-order and higher-order forms. Indeed, most dataflow languages are functional languages, and some researchers regard dataflow as a particular implementation technique for the functional paradigm (see the bibliography of [190] for a list of references on this subject). In particular, within functional programming STs are often referred to as being in *data passing form* and higher-order STs (third-order or above) are referred to as being in *agent passing form*.

As dataflow languages and functional languages are closely related, in some sense any functional programming language can be considered suitable for general purpose stream programming. For example, the well-known functional languages LISP, ML and MIRANDA (see [169] and [215]) can all be used to represent STs. However, whether such languages provide a natural

and straightforward mechanism for the specification of STs is less clear and for this reason several specialized stream orientated functional languages have been developed including ARTIC (see [60]), HOPE (see [52]) and RUTH (see [98]) designed to meet more specific needs such as real-time programming over streams.

4.2.1 Functional SPSs and Semantics.

Typically ADU-SPS and ANU-SPS are studied using the functional paradigm and as with dataflow languages the work of G Kahn has been widely adopted as a semantic approach for functional stream processing. However, other (sometimes related) approaches are also used including *greatest fixed points* (see [61] and [82]) and *Aczel's logical theory of constructions* (see [2], [71] and [72]). In addition, the work of [78] and [100] on *lazy evaluation* has provided an implementation technique for functional stream processing that has been widely adopted.

4.2.2 Applications.

The verification of functionally specified STs has been explored in the literature. In particular, operating systems have been an area of quite extensive research (see [119] for an overview) as the swapping of processes can be modelled using agent passing stream transformers. An example of operating system specification can be found in [46] and in addition this paper provides an example of how ANU-SPS can be specified using *classes* of functions.

As a more detailed example of functional stream processing research we now discuss the work of M Broy who has made a significant contribution to the development of techniques for functionally based stream processing. In particular, we discuss the FOCUS project that provides a functional framework for the specification of distributed systems based on stream communication.

The FOCUS Project. FOCUS (see [45]) is based on the work developed in [36], [39], [40], [41], [42], [47], [43], and [44].

FOCUS is not a language, but rather a collection of tools and modelling concepts that provide a framework for the description of parallel distributed systems as concurrent asynchronous processing elements. Within such networks data is exchanged via unbounded FIFO channels that are modelled as streams.

FOCUS aims to provide a theory of stepwise refinement and modular development of parallel systems and includes verification calculi that are intended to provide a formal system to reason about the correctness of system implementations at various level of abstraction. However, it is not the intention of FOCUS to provide a theory of stream based distributed processing (see [45]).

Despite the fact that FOCUS is a paradigm and not a language it does provide two concrete representations for expressing STs (as SPS). The first (and most abstract in the sense of specification) is the language AL based on AMPL (see [36]) and the second is the language PL based on the work in [47] and [62]. We discuss the languages AL and PL in Sections 9.9 and 9.10 respectively.

Given the specification of an ST in AL the FOCUS paradigm provides transformational rules (refinements) towards more concrete representations (in the sense of specification). Indeed, within FOCUS a representation is considered to be in its most concrete form (an implementation) if no further refinements and no further re-writings to another formalism (representation) are possible. Given this definition the implementation language of FOCUS can be consider to be PL, although one can imagine that these techniques could be extended to additional languages.

4.3 Logic Programming Languages with Streams

As logic programming provides a high-level and useful method of specification for some classes of systems it is natural that some researchers have explored the use of logic programming languages for the specification of SPSs. Indeed, there are several examples of modifications of relational languages for stream processing that can be found in the literature. In [179] these languages are divided into three groups:

(1) *Committed choice* parallel programming systems, for example, *PARLOG* (see [54]).

(2) Extension of PROLOG to include either the *parallel and* or *parallel or* operators (see for example [145]).

(3) Extension to PROLOG to include functional constructs, for example, [122], [146], [203], [174], [20], and [63].

However, a different classification can be found in [20] wherein logical languages for programming with streams are divided into two groups:

(A) Languages based on *static input-output mode variable declarations* for example the languages of [56] and [216].

(B) Languages based on *dynamic variable annotations* for example [55], [189] and [203].

While we are not aware of any work in the literature that describes the relationship between these two classifications, it is possible to make the following general comments on the methods used to incorporate the use of streams in logic programming.

4.3.1 Describing SPSs as Relations. The use of the term 'coroutine' in relational languages does not directly imply the use of streams (see [179]). However, typically logic programming languages modified for stream programming are designed to represent ADU-SPSs, although the particular description of ANU-SPSs will of course depend on the stream processing operations and types of concurrency allowed in the particular language.

4.3.2 The Use of Streams. As with the functional approach, streams are treated as the union of finite and infinite sequences. In particular, streams are typically implemented as finite lists, although the declaration and manipulation of infinite lists (and hence streams) may be permitted. However, the use of infinite lists in some relational languages may be non-terminating as they tend to use eager evaluation.

Specialized logic programming languages extended with non-strict processes and lazy evaluation to cope with stream programming are sometimes termed *perpetual processes* (see [147]) and have many similarities with functional languages. (A survey of the relationship between logical and functional languages can be found in [63] and [20].) Alternatively, relational languages can be modified to cope with streams by eliminating the *occurs check*, although this can lead to 'unsound inferences' (see [179]).

4.3.3 Semantics. Several semantic approaches have been adopted for dealing with perpetual processes including a fixed-point semantics in the style of Kahn. A discussion and comparison of these approaches can be found in [143].

4.3.4 Languages.

In addition to the specialized logic programming languages we have already mentioned in Section 4.3 we look in detail at a modification of PROLOG to cope with the use of streams.

5 Reactive Systems and Signal Processing Networks

The *reactive system* paradigm (see [91]) and *signal processing* paradigm are conceptually closely related. The essential difference between the two approaches is that reactive system research is concerned with SDB-SPSs and signal processing is concerned with SDU-SPSs; that is, in reactive systems channels are bidirectional. Consequently, from this point we will use the term 'reactive systems' to mean both reactive systems and signal processing networks.

Reactive systems are designed to model real-time systems such as operating systems and process control programs that 'repeatedly respond to inputs from their environment by producing outputs'. Stream communication provides a natural method for the specification of real-time systems. However, real-time system specification is not limited to this technique and is the reason that in general real-time system theory is less related to stream processing than the specialized real-time system research explored in reactive system theory. Therefore in this section we discuss reactive systems as a separate topic.

5.1 Streams, Signals and Sensors

Reactive systems and signal processing systems communicate via signals that are related to our concept of streams. Signals are divided into two types: *pure signals* that are un-typed and simply communicate an 'event' that can be used for synchronization; and typed signals that communicate data. Within the reactive system paradigm signals may be used for both input and output, but we note that typed signals are only used for input and are referred to as *sensors*. Given this informal definition signals in signal processing networks are all sensors. A comparison of typed signals and streams can be found in Section 9.7.

5.2 The Strong Synchrony Hypothesis and Multiform Time

The reactive system paradigm is based on what is referred to as either the *strong* or *perfect synchrony hypothesis* (see [26]) that requires all filters within a network to react instantly to input producing a corresponding output in zero time. As a consequence the whole computation performed by a reactive system is 'instantaneous'. In addition, reactive systems use what is referred to as a *multiform* notion of time (see [26]) wherein signals (streams) may be used as a time unit. As such, co-operation of sub-tasks (processes) defines new temporal relations that are used to define the global ordering of the data (compare [92]).

5.2.1 Semantics.

The semantics of reactive systems have been formalized using *temporal logic* (see [181]). In addition, [181] also includes a comparison of several different semantic approaches to general concurrent systems and how these approaches can be applied to reactive systems.

5.2.2 Languages.

In Sections 9.5, 9.7, and 9.8 we describe three languages for pro-

gramming reactive systems, respectively LUSTRE, SIGNAL, and ESTEREL, and contrast the different approaches that they take.

6 Stream Processing in the Design and Verification of Hardware

At many levels of abstractions of hardware description the role of *clocks* is an important one. The so-called *state transformer* formalization of hardware (see [94]) relies on the use of an abstract clock $T = \{0, 1, 2, \ldots\}$ to provide a discrete measure of the evolution within a device of the values of (for example) the registers and memory from some initial values to the values at some time $t \in T$ – referred to as the evolution of the device's *state*.

As the state transformer model of hardware can be naturally viewed as a special case of a stream transformer model of hardware, the use of streams is common in the hardware specification and hardware verification literature. Indeed, many of the stream processing techniques and languages that are discussed elsewhere in this survey are used for the study of hardware.

As such, most of the remaining literature on the subject of hardware specification and verification lies outside the scope of this paper as the use of streams is incidental rather than the main thrust of the research. (Although, the interested reader may still like to consult [81], [166], [58], [155], [224], [117], [232] and [89].) Therefore, in this section we discuss the topic of *synchronous concurrent algorithms*, a stream based computational model that has been used extensively for the study of hardware, but for technical and other reasons contrasts with many other approaches in this subject.

6.1 SCAs

The concept of a *synchronous concurrent algorithm* (SCA) was developed by B C Thompson and J V Tucker in the early 1980s (see [207], [206] and [208]), and was motivated originally by the need for an algebraic formalism for the specification and verification of general purpose hardware (see [94], [93], [92], [74], [96], [73] and [97] for case studies.) However, SCAs are also appropriate for the study of specialized hardware devices and specialized models of computation including: *systolic arrays* (see [207], [206], [104], [69] and [103]); *neural networks* (see [108], [109], [111], [230] and [210]); and *cellular automata and coupled map lattice dynamical systems* (see [149], [29], [110], [107] and [30]). (For general introductions to the topics of systolic architectures; neural networks; and cellular automata and coupled map lattice dynamical systems see respectively: [158], [137]; [154], [227], [171], [129], [130], [185], [186], [187]; [124] and [90]; and [219] and [229].)

Informally, an SCA can be visualized as a particular class of dataflow SDU-SPS; that is, a SCA is as a fixed, synchronous, deterministic dataflow network wherein modules compute and communicate in parallel via channels synchronized by a discrete global clock T. As such, all modules receive and produce data deterministically and hence the SCA as a whole also computes a total function. This is in contrast with all the other stream processing formalisms we have discussed that allow the specification of partial functions.

6.1.1 Streams.

Within SCA theory streams are represented using the function space $[T \to A]$ for some A of interest. However, again in contrast to the other approaches discussed, the overall functionality of a SCA is modelled using a *Cartesian form stream transformer* CFST

of the form

$$F^* : T \times [T \to A]^m \to A^n$$

rather than the more classical

$$F : [T \to A]^m \to [T \to A]^n.$$

While the use of CFST seems an apparently unimportant difference in specification technique (as F^* is essentially only the *un-Curried form* of F), Cartesian form specification is subtle in its implications. Specifically: from the perspective of computability the reconciliation of these two techniques is by no means straightforward (see [201]); and from the perspective of automated verification the use of Cartesian forms has significant advantages, in that it permits the use of essentially first-order techniques to establish the correctness of stream transformers (see [200]).

6.1.2 Semantics. A denotational semantics for SCAs is provided using *value functions* that are a special case of *primitive recursive functions* (see [209]). As such, SCAs have a rich theory founded in (generalized) computability theory, equational specification and term re-writing (see for example [22] and [200]) that addresses many theoretical issues that are neglected elsewhere in the stream processing literature.

The SCA computational model has also been generalized and formalized in several ways: *graph theoretical models* (see [159] and [165]); *process theoretic models* (see [211]); *operational semantic models* (see [206], [151], [150] and [182]); and *infinite SCAs* (see [153] and [152]).

6.1.3 Languages. Several formally equivalent languages have been developed for specifying, simulating and reasoning about SCAs: PR (see [206]); *FPIT* (see [206]), *CARESS* (see [150] and [182]) – both based on the concurrent assignment statement (see [225]); *ASTRAL*; and *PREQ* (see [200]). The language ASTRAL is discussed in Section 9.14.

7 Other Stream Processing Formalisms

7.1 ALPHA

While the language ALPHA (see [70]) is not specifically a stream processing language we mention ALPHA here as it is described by its authors as '...a grandson of Lucid...' and is used for the design and synthesis of systolic VLSI (see [218]). In particular, ALPHA is an equational language that involves a generalization of stream variables that can be used to represent a 'spatial domain'; that is, a (possibly infinite) matrix indexed by a sub-set of $\mathbb{Z}^n$. For example, if variable X is declared on the domain D defined by

$$D = \{(i, j) \mid i > 0, 1 \leq j \leq 2\}$$

then X represents a matrix

$$\begin{bmatrix} x_{1,1}, x_{1,2}, x_{1,3}, \ldots \\ x_{2,1}, x_{2,2}, x_{2,3}, \ldots \end{bmatrix}.$$

Using this methodology if variable Y was declared over domain D' defined by

$$D' = \{i \mid i > 0\}$$

then Y would essentially be a stream $y_0, y_1, y_2, y_3, \ldots$.

To compute over spatial domains ALPHA uses a generalization of point-wise extensions called 'motionless operators' whose semantics is formalized denotationally in the style of Kahn.

7.2 Stream X-Machines

Stream X-machines (see [106]) are based on the X-machine model of computation (a generalization of the Turing Machine – see [105]) that allow streams as both input and output. Stream X-machines have been used in the study of system testing and verification for which the authors claim they offer significant advantages.

8 Stream Processing Primitives and Constructs

8.1 Introduction

In this section, in order to clarify basic issues relating to computability we analyse in some detail the abstract stream processing primitives and constructs that can be found in the literature. In the following section we look at specific languages that are used to specify the particular classes of stream processing systems that we discussed in Sections 3, 4, 5 and 6.1.

8.2 Common Functional Stream Processing Operations

In this section we describe informally the typical functional stream processing primitives that can be found in the literature defined using a generalized concept of a stream. (However, we note in passing that these primitives are used in other formalisms as well, sometimes under a different name.) This formalization is based on the description given in [46].

In the sequel we use A to denote any data type (algebra), with sort names taken from the set $S \supseteq \{\mathbf{n}, \mathbf{b}\}$; that is, we assume that A includes as two of its carriers the natural numbers $\mathbb{N}$ (also denoted T when it is being used to represent discrete time) and the Booleans $\mathbb{B}$. A typical carrier of A is denoted A_s, for some $s \in S$, and A^w for some $w \in S^*$ is used to denote a Cartesian product $A_{s_1} \times A_{s_2} \times \cdots A_{s_n}$. For example, $A_{\mathbf{n}} = \mathbb{N} = T$ and $A_{\mathbf{b}} = \mathbb{B}$.

We denote a stream algebra by $\underline{A}$ with sort names taken from the set $\underline{S} = S \cup \{\underline{s} \mid s \in S\}$ with the convention that for each $s \in S$ that $\underline{A_s}$ is the stream $[T \to A_s]$.

In the following section we also assume that A is a continuous algebra with an appropriate partial ordering for each carrier. We use $A^\omega = [T \to A] \bigcup A^*$ to denote the set of all finite and infinite sequences (generalized streams) wherein $<> \in A^\omega$ denotes the empty sequence. A continuous mapping is denoted $\twoheadrightarrow$.

8.2.1 Functional Stream Processing Primitives.

(1) Stream construction operator. We define the *stream construction operator*, denoted **:**, with functionality **:** $: A \to A^\omega \twoheadrightarrow A^\omega$ by (in infix notation)

$$(\forall a \in A)\,(\forall s \in A^\omega) \quad a{:}s = s'$$

wherein if $|a| < |\mathbb{N}|$ then

$$(\forall t \in \{0, \ldots, |s|+1\}) \quad s'(t) = \begin{cases} a & \text{if } t = 0; \\ s(t-1) & \text{otherwise} \end{cases}$$

and if $|a| = |\mathbb{N}|$ then $s' = a$.

(2) Concatenation. We define the *concatenation operator*, denoted $\bullet$, with functionality $\bullet : A^\omega \to A^\omega \twoheadrightarrow A^\omega$ by (in infix notation)

$$(\forall a \in A^\omega) \quad <> \bullet a = a$$

and

$$(\forall (a{:}s), s' \in A^\omega) \quad (a{:}s) \bullet s' = a{:}(s \bullet s').$$

(3) First element selection. We define the *head operator*, denoted $\boldsymbol{hd}$ (and also $\boldsymbol{first}$), with functionality $\boldsymbol{hd} : A^\omega \to A^\perp$ by

$$\boldsymbol{hd}.<> = -$$

and

$$(\forall (a{:}s) \in A^\omega) \quad \boldsymbol{hd}.(a{:}s) = a.$$

(4) First element elimination. We define the *tail operator*, denoted $\boldsymbol{tl}$ (and also $\boldsymbol{rest}$), with functionality $\boldsymbol{tl} : A^\omega \to A^\omega$ by

$$\boldsymbol{tl}.<> = <>$$

and

$$(\forall (a{:}s) \in A^\omega) \quad \boldsymbol{tl}.(a{:}s) = s.$$

(6) Last element selection. We define the *last operator*, denoted $\boldsymbol{last}$, with functionality $\boldsymbol{last} : A^\omega \to A^\perp$ by

$$(\forall s \in A^\omega) \quad \boldsymbol{last}.s = \begin{cases} - & \text{if } s = <> \text{ or } |s| = |\mathbb{N}|; \\ a & \text{if } s = < a > \text{ for some } a \in A; \\ \boldsymbol{last}.(\boldsymbol{tail}.s) & \text{otherwise.} \end{cases}$$

(7) Filtering. We define the *filter operator*, denoted ©, with functionality $© : \wp(A) \to A^\omega \to A^\omega$ by (in infix notation)

$$(\forall S \in \wp(A)) \quad S© <> = <>$$

and

$$(\forall S \in \wp(A))\,(\forall (a{:}s) \in A^\omega) \quad S©(a{:}s) = \begin{cases} S©s & \text{if } a \notin S; \\ a{:}S©s & \text{otherwise.} \end{cases}$$

(8) Pointwise change. We define the *pointwise change operator*, denoted $.[. \mapsto .]$, with functionality $.[. \mapsto .] : A^\omega \to T \to A \to A^\omega$ by (in infix notation)

$$(\forall s \in A^\omega)\,(\forall t, t' \in T)\,(\forall a \in A) \quad s[t \mapsto a](t') = \begin{cases} s(t') & \text{if } t' \neq t; \\ a & \text{otherwise.} \end{cases}$$

It is also common in functional stream processing to use the following two higher-order primitives that act directly on STs themselves.

(A) After. We define the *after operation*, denoted $\ll$, with functionality $\ll : (A^\omega \twoheadrightarrow B^\omega) \to A \to (A^\omega \twoheadrightarrow B^\omega)$ by (in infix notation)

$$(\forall f \in (A^\omega \twoheadrightarrow B^\omega))\,(\forall a \in A)\,(\forall s \in A^\omega) \quad (f \ll a).s = f.(a{:}s).$$

(B) Then. We define the *then operation*, ambiguously denoted $\ll$, with functionality $\ll : B \to (A^\omega \twoheadrightarrow B^\omega) \to (A^\omega \twoheadrightarrow B^\omega)$ by (in infix notation)

$$(\forall b \in B)\,(\forall f \in (A^\omega \twoheadrightarrow B^\omega))\,(\forall s \in A^\omega) \quad (b \ll f).s = b{:}f.s.$$

8.3 Stream Processing Primitives in Logic Programming

In this section we identify four generic stream processing primitives that can be found in the logic programming literature. We conclude the section with some concrete examples of these types of stream processing primitives based on the list given in [17].

8.3.1 Generic Relational Stream Processing Primitives. In [179] stream processing primitives in logic programming languages are referred to as *transducers* (see [1]) and are divided into four groups. However, as pointed out in [178] this list of transducer types is not exhaustive, although no indication is given as to why this is the case.

(1) Enumerators (Generators). Enumerators produce a stream derived from some initial values. A generic enumerator definition is as follows:

```
enumerate(Stream) :-
    initial_state(State),
    enumerate(State,Stream).

enumerate(S,[X | Xs]) :-
    next_state_and_value(S,NS,X),
    !,
    enumerate(NS,Xs).

enumerate(_,[]).
```

(2) Maps. Maps produce an output stream by applying a function to an input stream. A generic map definition is as follows:

```
map_f([X | Xs],[Y | Ys]) :-
    f(X,Y),
    map_f(Xs,Ys).

map_f([],[]).
```

(3) Filters. Filters produce part of their input stream as output, the elements selected being based on defined criteria. A generic filter definition is as follows:

```
filter([X | Xs],Ys) :-
    inadmissible(X),
    !,
    filter(Xs, Ys).

filter([X | Xs],[X | Ys]) :-
    filter(Xs,Ys).

filter([],[]).
```

(4) Accumulators. Accumulators produce an 'aggregate' of input values as output. A generic enumerator definition is as follows:

```
accumulate(Stream,Value) :-
    initial_state(State),
    accumulate(List,State,Value).

accumulate([X | Xs],S,Value) :-
    next_state(X,S,NS),
    accumulate(Xs,NS,Value).

accumulate([],S,Value) :-
    final_state_value(S,Value).
```

Notice that accumulators are strictly first-order primitives.

8.3.2 Examples of Relational Stream Processing Primitives.

We now list the examples of second-order stream processing primitives (in functional form) presented informally in [179] based on the list given in [17].

(A) For each constant $c \in A_s$ for some $s \in S$ we define

$$ConStr^c :\rightarrow [T \rightarrow A_s]$$

by

$$(\forall t \in T) \quad ConStr^c(t) = c.$$

(B) For each constant $i \in \mathbb{Z}$ we define

$$IntFrom^i :\to [T \to \mathbb{Z}]$$

by

$$(\forall t \in T) \quad IntFrom^i(t) = i + (t - 1).$$

(C) For each binary operator $\sigma : A_s \times A_s \to A_s$ for some $s \in S$ we define

$$Agg^\sigma : [T \to A_s] \to [T \to A_s]$$

by

$$(\forall a \in [T \to A_s])\,(\forall t \in T) \quad Agg^\sigma(a)(t) = \begin{cases} a(0) & \text{if } t = 0\text{, and} \\ \sigma(Agg^\sigma(a)(t-1), a(t)) & \text{otherwise.} \end{cases}$$

(D) For each unary operator $\sigma : A_s \to A_s$ for some $s \in S$ we define

$$Map^\sigma : [T \to A_s] \to [T \to A_s]$$

by

$$(\forall a \in [T \to A_s])\,(\forall t \in T) \quad Map^\sigma(a)(t) = \sigma(a(t)).$$

(E) For each binary relation $\rho \subseteq A_s \times A_s$ for some $s \in S$ we define

$$Com^\rho : [T \to A_s] \times [T \to A_s] \to [T \to \mathbb{B}]$$

by

$$(\forall a_1, a_2 \in [T \to A_s])\,(\forall t \in T) \quad Com^\rho(a_1, a_2)(t) = \begin{cases} tt & \text{if } \rho(a_1(t), a_2(t))\text{, and} \\ f\!f & \text{otherwise.} \end{cases}$$

(F) For each $n \in \mathbb{N}$ and for each $s \in S$ we define

$$Rep^n_s : [T \to A_s] \to [T \to A_s]$$

by

$$(\forall a \in [T \to A_s])\,(\forall t \in T) \quad Rep^n_s(a)(t) = a(t \; div \; n).$$

(G) For each $s \in S$ and for each $n, x \in A_s$ we define

$$Lag^{n,x}_s : [T \to A_s] \to [T \to A_s]$$

by

$$(\forall a \in [T \to \mathbb{N}])\,(\forall t \in T) \quad Lag^{n,x}_s(a)(t) = \begin{cases} x & \text{if } t < n\text{, and} \\ a(t-n) & \text{otherwise.} \end{cases}$$

(H) For each $s \in S$ we define

$$Merge_s : [T \to A_s] \times [T \to A_s] \to [T \to A_s]$$

by

$$(\forall a_1, a_2 \in [T \to A_s])\,(\forall t \in T) \quad Merge_s(a_1, a_2)(t) \begin{cases} a_1(t) & \text{if } t \text{ is even, and} \\ a_2(t) & \text{otherwise.} \end{cases}$$

9 Stream Processing Languages

As promised we now examine some examples of stream processing languages designed to represent the particular classes of SPSs we have identified in the literature.

In Section 9.4 and Section 9.5 we discuss the languages Lucid and LUSTRE designed to programme asynchronous and synchronous dataflow SPSs respectively. Also, in Section 9.6 we briefly discuss the so-called Manchester Languages and mention some other dataflow languages that can be found in the literature.

In Section 9.7 and Section 9.8 we discuss the related languages SIGNAL and ESTEREL that are used for programming signal processing networks and reactive systems respectively.

In Sections 9.9, 9.10 and 9.11 we discuss the functional languages AL, PL and Daisy respectively.

In Section 9.12 we examine a modification of PROLOG designed for stream programming.

In Section 9.13 we look at the language STREAM used in the design and verification of hardware.

Finally, in Section 9.14 we describe the language ASTRAL that has been developed from SCA theory. However, we begin this section with a discussion of the RS-Flip-Flop, that we will use as a running example for presenting and hence comparing the syntax of the stream processing languages that we discuss. We note that we choose to use a running example combined with a formal analysis of language constructs, rather than use specific examples tailored to demonstrate the features of each language, as we believe this is more objective and more in keeping with the aim of this survey as discussed in the introduction. The reader interested in examples that have motivated specific features of our example languages is directed to the references cited in the appropriate section.

9.1 A Running Example: the RS-Flip-Flop

The RS-Flip-Flop (or simply Flip-Flop) is a widely occurring device found in computer hardware. The Flip-Flop is designed to output a stream of 'true' (tt) and 'false' (ff) signals controlled by two input streams of true and false *control signals*.

Valid control signals consist of one of three simultaneous input pairs:

- 'Reset' – (tt,ff). This indicates that the Flip-Flop's next output should be a ff.

- 'Set' – (ff,tt). This indicates that the Flip-Flop's next output should be a tt.

- 'Hold' – (ff,ff). This indicates that the Flip-Flop should repeat its previous output.

However, while the pair (tt,tt) is considered to be illegal input, a practical implementation of the Flip-Flop must be able to cope with this input.

9.2 Formalization of the Flip-Flop as a ST

This informal description of the Flip-Flop's operation can be made more precise by defining the Flip-Flop as an abstract ST as follows:

$$\text{Flip-Flop} : [T \to \mathbb{B}]^2 \to [T \to \mathbb{B}]$$

defined by

$$(\forall b_1, b_2 \in [T \to \mathbb{B}]) \quad \text{Flip-Flop}(b_1, b_2)(0) = tt$$

and

$$(\forall b_1, b_2 \in [T \to \mathbb{B}])\, (\forall t \in T)$$

$$\text{Flip-Flop}(b_1, b_2)(t+1) = \begin{cases} \mathit{ff} & \text{if } b_1(t) = tt \text{ and } b_2(t) = \mathit{ff}; \\ tt & \text{if } b_1(t) = \mathit{ff} \text{ and } b_2(t) = tt; \text{ and} \\ \text{Flip-Flop}(b_1, b_2)(t) & \text{otherwise.} \end{cases}$$

In particular, notice that this specification outputs its previous output if the illegal control signal (tt,tt) is supplied as input.

9.3 An Implementation of the Flip-Flop as a SPS

A typical implementation of the Flip-Flop can be visualized at the conceptual level as a SDU-SPS comprising two input streams, two modules, and two output streams wherein both modules compute the 'nor' function. To reconcile this implementation with the functionality of the specification only one stream is considered as 'proper output' (the first module's output), with the other stream used only as 'feedback' to compute the Flip-Flop's next output.

9.3.1 The Flip-Flop SPS's Computation. The SPS representing the Flip-Flop is shown in Figure 2. Initially the modules of the SPS representing the Flip-Flop will output

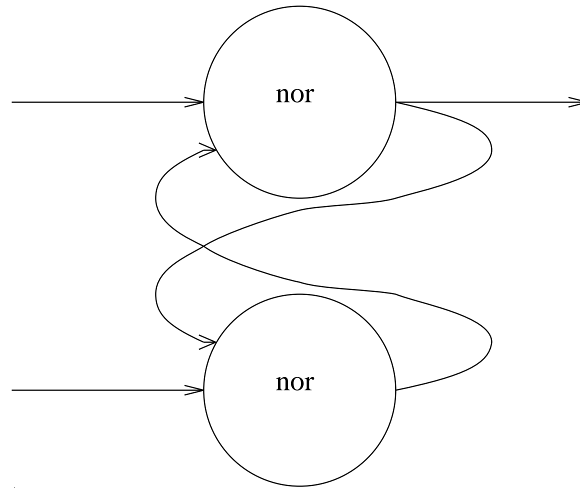

Figure 2: The RS-Flip-Flop as a SPS

some initial values that for convenience we will assume is the pair (tt,ff).

After the Flip-Flop's initial output each module computes (synchronously) on the streams of control signals and the previous output of the other module to produce the next output.

9.3.2 Properties of the Flip-Flop. We note that this description of the Flip-Flop is a highly conceptualized model of idealized hardware, for which the Flip-Flop SPS requires pre-

and post-processing of its input and output respectively to meet its specification. Indeed, it has been pointed out that 'real' discrete models of Flip-Flop implementations should be developed using three- or four-valued logic (see for example [48], [228] and [133]). This stems from the fact that associating *tt* and *ff* with low and high signal values leads to a false correspondence between boolean logic and voltage values.

However, as our use of this example is only to demonstrate certain basic, but important language features we find it convenient. In particular, our model of the Flip-Flop enables us to show how mutual recursion is expressed; how vector-valued components are specified; and how each language deals with explicit synchronization.

The interested reader can find studies of the Flip-Flop at various levels of abstraction in [114], [208], [200] and [134].

9.4 Lucid

Lucid ([223]) is perhaps the best known of all the dataflow languages that have been developed. A Lucid programme is essentially a system of recursion equations, although Lucid is described by its authors as a 'functional dataflow programming language'. The term 'dataflow' is chosen because each Lucid programme is semantically equivalent to a dataflow network; and 'functional' because the output of each filter is a function of its inputs. (Note that the term 'functional' used here does not imply a computation without side-effects as in the mathematical sense.) Lucid is also described by its authors as a 'typeless' language as there is no declaration section. However, a more formal description would be to say that Lucid operators are overloaded and their type is inferred from their context.

Lucid was conceived by its authors in 1974 with what they claim to be quite modest aims; that is, to show that real-life programmes could be written in a purely declarative style so that programme verification would be possible. The authors felt that a purely functional language was not creditable for this purpose for reason of efficiency, and so Lucid contains iterative constructs so that (the authors claim) when writing Lucid programmes the programmer may make use of algorithms used in real 'everyday' programming. It was also (later?) intended that Lucid could exploit the new highly-parallel, multiprocessor dataflow machines.

9.4.1 Constructs and Primitives. Each Lucid programme is an expression structured using the 'where' clause taken from *ISWIM* (see [140]) over simple 'data types', for example: integers; reals; Booleans; words; character strings; and finite lists. Lucid also uses the *if...then...else* construct.

Lucid has the 'usual' operators over the data types just mentioned and treats them as pointwise extensions over time and hence can be used to manipulate streams directly. In addition Lucid uses six explicit stream processing primitives with the following semantics.

(1) First. For each $\underline{u} \in \underline{S}^+$ we define

$$First^A_{\underline{u}} : [T \to A^u] \to [T \to A^u]$$

by

$$(\forall a \in [T \to A^u])\,(\forall t \in T) \qquad First^A_{\underline{u}}(a)(t) = a(0).$$

(2) Next. For each $\underline{u} \in \underline{S}^+$ we define

$$Next^A_{\underline{u}} : [T \to A^u] \to [T \to A^u]$$

by

$$(\forall a \in [T \to A^u])\,(\forall t \in T) \quad Next^A_{\underline{u}}(a)(t) = a(t+1).$$

(3) Followed By. For each $\underline{u} \in \underline{S}^+$ we define

$$Fby^A_{\underline{u}} : [T \to A^u] \times [T \to A^u] \to [T \to A^u]$$

by

$$(\forall a_1, a_2 \in [T \to A^u])\,(\forall t \in T) \quad Fby^A_{\underline{u}}(a_1, a_2)(t) = \begin{cases} a_1(t) & \text{if } t = 0, \text{ and} \\ a_2(t-1) & \text{otherwise.} \end{cases}$$

(4) At Time. For each $\underline{u} \in \underline{S}^+$ we define

$$AtTime^A_{\underline{u}} : [T \to A^u] \times [T \to \mathbb{N}] \to [T \to A^u]$$

by

$$(\forall a \in [T \to A^u])\,(\forall n \in [T \to \mathbb{N}])\,(\forall t \in T) \quad AtTime^A_{\underline{u}}(a, n)(t) = a(n(t)).$$

(5) Whenever. For each $\underline{u} \in \underline{S}^+$ we define

$$Whenever^A_{\underline{u}} : [T \to A^u] \times [T \to \mathbb{B}] \to [T \to A^u]$$

by

$$(\forall a \in [T \to A^u])\,(\forall b \in [T \to \mathbb{B}])\,(\forall t \in T)$$

$$Whenever^A_{\underline{u}}(a, b)(t) = \begin{cases} a(t) & \text{if } b(t) = tt, \text{ and} \\ Whenever^A_{\underline{u}}(a, b)(t+1) & \text{otherwise.} \end{cases}$$

(6) As Soon As. For each $\underline{u} \in \underline{S}^+$ we define

$$Asa^A_{\underline{u}} : [T \to A^u] \times [T \to \mathbb{B}] \rightsquigarrow [T \to A^u]$$

by

$$(\forall a \in [T \to A^u])\,(\forall b \in [T \to \mathbb{B}])\,(\forall t \in T) \quad Asa^A_{\underline{u}}(a, b)(t) = a(\mu\, k.[b(k) = tt]).$$

(7) Upon. For each $\underline{u} \in \underline{S}^+$ we define

$$Upon_{\underline{u}}^{A} : [T \to A^u] \times [T \to \mathbb{B}] \to [T \to A^u]$$

by

$$(\forall a \in [T \to A^u])\,(\forall b \in [T \to \mathbb{B}])\,(\forall t \in T)$$

$$Upon_{\underline{u}}(a,b)(t) = \begin{cases} a(0) & \text{if } t = 0, \\ a(\text{NumOfTrues}(Next_{\underline{\mathbf{b}}}^{A}(b))(t)) & \text{otherwise} \end{cases}$$

wherein $Next_{\underline{\mathbf{b}}}^{A}$ is defined as above and NumOfTrues : $[T \to \mathbb{B}] \to [T \to \mathbb{N}]$ is defined by

$$(\forall b \in [T \to \mathbb{B}])$$

$$\text{NumOfTrues}(b)(0) = \begin{cases} 1 & \text{if } b(0) = tt, \text{ and} \\ 0 & \text{otherwise} \end{cases}$$

and

$$\text{NumOfTrues}(b)(t+1) = \begin{cases} 1 + \text{NumOfTrues}(b)(t) & \text{if } b(t+1) = tt, \text{ and} \\ \text{NumOfTrues}(b)(t) & \text{otherwise.} \end{cases}$$

9.4.2 The use of Streams.

As with many of the other languages we will discuss streams are represented as variables. In the particular case of Lucid any free variables (not explicitly declared) are treated as input streams.

9.4.3 Language Development and Current Uses.

Since its conception various implementations of Lucid have been written (see [75] and [188]) and one such implementation *pLucid* – Lucid over the algebra of POP-2 taken from [51] – has been used experimentally for software design (see [221]). Recently work on GLU (Granular Lucid) has appeared in [11].

9.4.4 Lucid Syntax.

The RS-Flip-Flop can be described in Lucid as follows:

```
flipflop(In1, In2) = (Out1, Out2)
    where
        Out1 = true fby (In1 nor Out2)
        Out2 = false fby (In2 nor Out1)
```

9.5 LUSTRE

LUSTRE (see [53]) is a synchronous dataflow language related to Lucid. Like Lucid it is based on the description of a SPS as a system of equations. However, unlike Lucid, LUSTRE requires that the output at time t of the functions defined by such a set of equations depends only on input received either before or at time t. This property is referred to by the authors of LUSTRE as *causality*.

We note in passing that intuitively *causality* appears to restrict LUSTRE to expressing the

class of *course-of-values recursive functions* (see [212]). However, the authors do not discuss the issue of computability in this respect.

In common with languages for describing reactive systems LUSTRE is based on the *strong synchrony hypothesis* and has a *multiform* notion of time (see Section 5). Furthermore, in common with the language ESTEREL (see Section 9.8) LUSTRE programs are implemented via compilation into finite automata.

The authors state that LUSTRE programs are subject to a strict analysis for *deadlock* based on a domain theoretic analysis of the various clocks defined using the *When* operator, rather than by the *cycle sum test* that is applied to Lucid programmes (see [222]). However, the authors concede that while this approach does detect any potential deadlock it also rejects some valid programmes. It is this strict approach to the interplay between the various clocks over which the various filters compute within a programme that ensures the synchronous nature of LUSTRE.

9.5.1 Primitives and Constructs. In common with Lucid underlying operations are treated as point-wise extensions over time in LUSTRE and can be directly applied to streams.

Any LUSTRE program, that is correct with respect to the various static-semantic tests that are applied to it, is compiled into a simplified basic abstract syntax. Compilation into this restricted syntax eliminates separate node (filter) definitions, used to employ a modular programming technique. In particular, stream operators are compiled into a restricted subset of stream operators that form a functionally complete set. This functionally complete set consists of the following four operations that we now define informally.

Let A be some algebra wherein $S = \{s_1, \ldots, s_n\}$ for some $n \in \mathbb{N}$. Also, let $\mathbb{U} =< \mathbb{U}_{s_1}, \ldots, \mathbb{U}_{s_n} >$ be some collection of distinct values such that $\mathbb{U}_{s_i} \notin A_{s_i}$ for $i = 1, \ldots, n$, and let $A^{\mathbb{U}} = A \cup \mathbb{U}$.

(1) Previous. For each $\underline{u} \in \underline{S}$ we define

$$Pre^A_{\underline{u}} : [T \to A]^u \to [T \to A^{\mathbb{U}}]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall t \in T) \quad Pre^A_{\underline{u}}(a)(t) = \begin{cases} (a)(t-1) & \text{if } t > 0, \text{ and} \\ (\mathbb{U}_{u_1}, \ldots, \mathbb{U}_{u_{|u|}}) & \text{otherwise.} \end{cases}$$

(2) Followed By. For each $\underline{u} \in \underline{S}$ we define

$$FBy^A_{\underline{u}} : [T \to A]^u \times [T \to A]^u \to [T \to A]^u$$

by

$$(\forall a_1, a_2 \in [T \to A]^u)\,(\forall t \in T) \quad FBy^A_{\underline{u}}(a_1, a_2)(t) =$$

$$\begin{cases} a_1(0) & \text{if } t = 0, \text{ and} \\ a_2(t) & \text{otherwise.} \end{cases}$$

Notice that this is different from the Lucid operator *Fby*.

(3) When. For each $\underline{u} \in \underline{S}$ we define

$$When^A_{\underline{u}} : [T \to A]^u \times [T \to \mathbb{B}] \rightsquigarrow [T \to A]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall b \in [T \to \mathbb{B}])\,(\forall t \in T) \quad When^A_{\underline{u}}(a,b)(t) = a(\mu\, k.[b(k) = tt \wedge k \geq t]).$$

(4) Current. For each $\underline{u} \in \underline{S}$ we define

$$Current^A_{\underline{u}} : [T \to A]^u \times [T \to \mathbb{B}] \to [T \to A]^u$$

by

$$(\forall a \in [T \to A]^u)\,(\forall b \in [T \to \mathbb{B}])\,(\forall t \in T)$$

$$Current^A_{\underline{u}}(a,b)(t) = \begin{cases} a(t) & \text{if } b(t) = tt \\ Current^A_{\underline{u}}(a,b)(t-1) & \text{if } b(t) = ff \wedge t > 0 \\ When^A_{\underline{u}}(a,b)(0) & \text{otherwise.} \end{cases}$$

9.5.2 The use of Streams. As is common in equational stream processing languages undefined variables are treated as input streams in LUSTRE programmes. Indeed, the notion of a stream in LUSTRE is the same as in standard dataflow and is not the same as in the reactive system paradigm. It is for this reason that we choose to classify LUSTRE as a dataflow language.

9.5.3 Semantics. Two separate approaches to the semantics of LUSTRE have been applied. The first is a domain-theoretic approach in the style of Kahn's work. The second approach is an operational semantics based on the work of Plotkin (see [180]). This operational semantics can been used for proofs of equivalence of different LUSTRE programs, and is the semantic model that has been used to analyse the properties of the compilation of LUSTRE programmes into finite automata.

9.5.4 Language Development and Current Uses. LUSTRE has been used for such diverse applications as music synthesis description (see [8]) and for verification of real-time systems (see [87]).

9.5.5 Syntax. The RS-Flip-Flop can be expressed in LUSTRE as follows:

```
node flipflop(In1, In2 : bool)
    returns(Out1, Out2 : bool);
let
    Out1 = tt : FBy pre(In1) nor pre(Out2);
    Out2 = ff : FBy pre(In2) nor pre(Out1);
tel
```

9.6 Other Dataflow Languages

9.6.1 The 'Manchester Languages'. There are several so-called 'Manchester Languages' (see [101]) including SASL, SISAL, LAPSE and MAD that have been used on the Manchester Dataflow Machine. In this Section we very briefly discuss these languages. The reader interested in the topic of specialized dataflow architecture can consult [86] and more recently [190].

SASL. The language SASL (see [101]) is a functional language. SASL derives its name from the fact that only single assignment functions (one argument) are permitted. Multiple argument functions are achieved with *Currying*.

SISAL. The language SISAL (see [156]) is a typed 'value orientated' functional language designed for dataflow computing machines. The name SISAL is derived from Streams and Iteration in a Single Assignment Language. SISAL allows recursive constructs and looping. In addition to being implemented on the Manchester Machine, SISAL has also been implemented on the VAX, CRAY and HP dataflow machines (see [190]).

VALID. The language VALID (see [7]) is a higher-order functional language designed to achieve very high-level parallelism. VALID derives its name from Value Identification Language and has a mix of ALGOL- and LISP-like syntax, including block-structuring and case statements.

DCBL. The language DCBL (pronounced 'decibel' – see [101]) is a high-level dataflow language designed to define the operational semantics for dataflow computing languages. In particular, DCBL is designed to enable users to express programmes with many forms of concurrency, at a high-level of abstraction without any machine dependent characteristics.

9.6.2 General Dataflow Languages.

VAL. The language VAL (see [68] and [33]) is a synchronous functional language with implicit concurrency. The name VAL is derived from the languages 'value orientated' rather than 'variable orientation' nature; that is, new values can be derived, but cannot be modified. This principle is used in the language so that values can be assigned to identifiers, but identifiers cannot be used as variables in order to address certain issues arising from the automatic generation of concurrent implementations.

ID. The language ID (see [9]) is an un-typed, functional, block-structured language that supports non-determinism and the use of streams. A programme in ID consists of a list of expressions wherein each expression is either a 'loop', a 'conditional', a 'block' or a 'procedure application'.

9.7 SIGNAL

SIGNAL (see [84]) is an applicative language designed to programme real-time systems using synchronous dataflow. The authors claim that a SIGNAL representation is very close to the specification of a system, either mathematical or graphical, and leads to an elegant formal

'synchronization calculus'.

SIGNAL uses two concepts of time: *logical time* and an associated timing calculus based on the strong synchrony hypothesis (see Section 5); and *physical time*. Using this system temporal references are determined entirely by the sequence of communication events and not as (the authors claim) by the input events as in either LUSTRE or the dataflow approach.

Individual processing elements in a SPSs described by SIGNAL are *not* synchronized by a single global clock $T = \{0, 1, 2, \ldots\}$, rather SIGNAL has a 'multiform notion' of time (see Section 5).

9.7.1 The use of Streams.

The name SIGNAL is derived from the infinite sequences called *signals* over which all processes in a SIGNAL system compute (see Section 5.1). Each signal is a map $a : T \to A$ for some data set A and some *clock* $T = \{1, 2, \ldots\}$. (Notice that the clock starts at 1 and not 0.) It would appear from this description that signals are streams. However, the individual values of a signal may be 'sampled' at continuous points rather than simply at the discrete division indicated by the signal's clock. In addition, the values are not persistent and as such may only be sampled in order; that is, once the value of a signal a has been sampled at time $t \in T$ it may henceforth only be sampled at some time t' wherein $t' > t$. (Also see the following section on further operators.) Notice that this interpretation of a signal is related to Kahn's visualization of streams as asynchronous FIFO queues (see Section 3.3.1).

9.7.2 Constructs and Primitives.

SIGNAL operators are divided into two classes: 'S-operators' that define signals and 'P-operators' that are used to create interconnections between processes. We will only consider signal definition operators here.

(1) Basic Operations. The syntax

$$a := b + 1$$

for some signals $a \in [T \to A]$ and $b \in [T' \to A]$ for some data set A wherein 1 is a constant signal creates a process with the following semantics:

$$(\forall t \in T) \qquad a(t) = b(t) + 1;$$

that is, it creates a process that takes a single signal input b and produces a single signal output a that at every time cycle t is precisely the value of $a(t)$ plus one.

Notice here that because of the nature of the process specified the two clocks T and T' are synchronized and hence considered to be the same. This is not a property of signal processes in general.

(2) Delays. The syntax

$$a \ init \ c$$
$$a := b \,\$1$$

for some signals $a \in [T \to A]$ and $b \in [T' \to A]$ for some data set A wherein c is a constant signal and creates a process with the following semantics:

$$(\forall t \in T) \qquad a(t) = \begin{cases} c & \text{if } t = 0 \text{ and} \\ b(t-1) & \text{otherwise;} \end{cases}$$

that is, the statement creates a process with a single input that delays its output by one time cycle and outputs a constant at time $t = 0$.

Notice here that a delay is defined by two separate processes (statements) and hence if the first statement is omitted (as in some of the reference examples) then the signal described by its process is undefined at time $t = 0$. Also, there is an inconsistency in the reference examined in that the signal's underlying clocks are given as $T = \{1, 2, \ldots\}$, but the *init* statement define values of streams at time $t = 0$.

(3) Composition. The syntax

$$(|a\ init\ c|a := b + 1|b := a\ \$1|)$$

denotes the process formed by the composition of the processes $a\ init\ c$, $a := b + 1$ and $b := a\ \$1$ specified in the previous examples. The ordering of the sub-processes within a composition is unimportant; that is, it is associative and commutative, and communication is implied between processes wherein an output signal of one process (an identifier on the left of an ':=') has the same name as an input signal (an identifier on the right of an ':=') from a *different* process. So our example has the intended semantics

$$(\forall t \in T) \quad a(t) = \begin{cases} c & \text{if } t = 0 \text{ and} \\ a(t-1) + 1 & \text{otherwise;} \end{cases}$$

(4) Further operators. SIGNAL also uses the operators *when*, *event* and *synchro* with the following syntax

$$a := b\ when\ c,$$

$$a := event\ b$$

and

$$synchro\ a, b$$

respectively. Because the semantics of these statements is 'formalized' using a *clock calculus*, that we will not discuss, we will only give the intuitive meanings of these statements: *when* is a so-called *undersampling* operator that, in the context of our example, produces the input signal b if it is defined at the same time the Boolean signal c is defined and 'true'; *event* delivers an always 'true' Boolean signal whenever (in the context of our example) signal b is defined; and *synchro* (again in the context of our example) explicitly synchronizes the signal's a and bs clocks.

Because of the lack of a global clock and the definition of a signal, when examining the current value of a particular signal $a : T \to A$ we have two possible results: it may either be undefined or will have some value in the data set A. Because of this definition of signals the authors use a *clock calculus* to give and check the semantics of SIGNAL definitions. As Boolean signals are used to define clocks (via the *event* operator) this clock calculus requires two data sets (and is the reason, the authors claim, that a Boolean calculus is insufficient); that is, $C = \{-1, 0, 1\}$ for Boolean signals, wherein 0 denotes the absence of a value, -1 denotes 'false', and 1 denotes 'true'; and $C' = \{0, 1\}$ for all other

signals, wherein 0 denotes the absence of a value and 1 the presence of a value. Within this calculus the data set C is given the structure of a commutative field onto which all SIGNAL processes can be mapped. This 'mapping' of a process is used to analyse the relationship of any sub-modules clocks and to detect incorrectly defined processes. For example, the compositional process

$$(|x := \ a \ when \ (a > 0)|y := a \ when \ (not(a > 0))|z := x + y|)$$

gives rise to the following equations in the clock calculus (using c to represent the Boolean expression $a > 0$)

$$x^2 = a^2(-c - c^2)$$
$$y^2 = a^2(c - c^2)$$
$$z^2 = x^2 = y^2$$

that gives $-c = c$. As this has a single solution ($c = 0$) the process defined by this composition is undefined. This is intuitively clear from the example as the clocks over which x and y are defined are mutually exclusive.

9.7.3 Semantics.

SIGNALS semantics is based on the *clock calculus* described above that we will not discuss further.

9.7.4 Syntax.

The RS-Flip-Flop can be expressed as follows in SIGNAL:

(| *Out1* ***init*** *tt* | *Out2* ***init*** *ff* |
In1′ := *In1$1* | *In2′* := *In2$1* |
Out1′ := *Out1$1* | *Out2′* := *Out2$1* |
Out1 := *In1′* nor *Out2′* | *Out2* := *In2′* nor *Out1′*
|*)*

9.8 ESTEREL

ESTEREL (see [24], [25], [26] and [32]) is a real-time imperative concurrent language for describing reactive systems. However, ESTEREL is designed for describing ADB-SPSs rather than ADU-SPSs as in the case of the languages LUSTRE and SIGNAL. (See the following section on the use of streams in ESTEREL.)

The authors state that the aim of ESTEREL is to develop a rigorous formal model of real-time computation with an operational semantics that can be used for tasks where programming using conventional languages is difficult.

9.8.1 Constructs and Primitives.

The basic structuring device in an ESTEREL programme is the *module* with input and output signals for broadcast communication and internal signals for internal broadcast communication.

The body of a module that describes its operation can include the following basic primitives and constructs:

(1) Null process. The command

nothing

creates a process that does nothing in zero time.

(2) Local variable declaration. The command

var X : *type* in i end

creates a local variable X for process i.

(3) Variable assignment. The command

X := *exp*

assigns variable X with the value of the expression *exp*.

(4) Signal Transmission. The command

emit s(*exp)*

emits the value of *exp* on signal s.

(5) Conditional execution. The command

do i upto s(*exp*)

repeatedly execute process i until the value *exp* is broadcast onto signal s and

do i uptonext s(*exp*)

repeatedly execute process i until the value *exp* is broadcast onto signal 's' twice.

(6) Sequential Composition. The command

$$i_1;\ i_2$$

invokes process i_2 immediately upon completion of process i_1.

(7) Parallel Composition. The command

$$i_1 || i_2$$

simultaneously invokes processes i_1 and i_2 sharing the same local variables and local signals.

(8) Iteration. The command

loop i end

executes process i in a continuous loop. However, processes like

X := 0; loop X := X + 1 end; loop emit s(X) end

have no semantics, due to the strong synchrony hypothesis, and are checked for during static semantic evaluation.

(10) If Then Else. The command

if *boolexp* then i_1 else i_2 fi

has the usual semantics, but because of the strong synchrony hypothesis, we assume here that *boolexp* is evaluated in zero time so control is passed immediately to either i_1 or i_2.

(11) Process termination. The command

tag T in i end
exit T

executes process i until 'exit T' is executed (in i) whereupon process i is terminated.

From these basic primitives many 'higher-level' construct are formed. However, these are just for convenience during programming and do not occur in the semantic model.

9.8.2 The use of Streams. ESTEREL uses the same notion of stream processing as SIGNAL; that is, signals and a multiform notion of time. However, unlike SIGNAL in ESTEREL some signals are used for both input and output from processes and information is broadcast in the sense that complete connectivity is assumed between processes. A commutative operator '*' is explicitly associated with each signal to deal with simultaneous transmission (see [168]) such that if the values $v_1, v_2, \ldots, v_n$ for $n > 1$ are broadcast simultaneously onto a signal s then the value on s is $v_1 * v_2 * \cdots * v_n$.

9.8.3 Semantics. ESTEREL has a complicated semantic model with three different levels:

(1) Static Semantics. Used to establish temporal relations between processes and check for any temporal paradoxes.

(2) **Behavioural Semantics.** Used to define the temporal behaviour with respect to the static semantics.

(3) **Computational Semantics.** Used to establish exactly what a program computes.

Once the computational semantics has been established any concurrency is eliminated by compiling into a sequential programme that is implemented as an automaton in *C* (for example) by a similar method used in parser generators (see for example [204]). The authors are confident that this technique leads to an efficient implementation.

9.8.4 Language Development and Current Uses. ESTEREL has been used for HCI and for programming communication protocols and real-time controllers (see [57], [173] and [27] respectively). An ESTEREL environment exists (see [31]) that includes simulators, debugging tools and a compiler to hardware, based on the techniques discussed in [23]. One current research aim is to implement existing ESTEREL programmes directly in hardware.

9.8.5 Syntax. The RS-Flip-Flop can be described in ESTEREL as follows:

```
var L1,L2 : bool in flipflop ;
module flipflop:
    input In1, In2 : bool ;
    output Out1, Out2 : bool ;
    L1 := true ;
    L2 := false ;
    emit Out1(L1) ;
    emit Out2(L2) ;
    loop
        L1 := In1 nor L2 ;
        L2 := In2 nor L1 ;
        emit Out1(L1) ;
        emit Out2(L2) ;
    end.
```

9.9 AL

AL is a typed equational language that provides a specification formalism for (potentially) recursive stream operations. Implicit concurrency is expressed by the juxtaposition of equational definitions within both programme and agent definitions.

9.9.1 Constructs and Primitives. AL uses a block structure and includes constructs such as *if...then...else...fi*. It also includes the finite choice operator $\Box$, and hence is AL is able to define non-deterministic behaviour.

AL has all of the basic stream processing primitives as described in Section 8.2 as built in operators. In addition, *functions* mapping data to data and *components* mapping data and streams of data to streams of data can be defined by the user.

9.9.2 The use of Streams.

The declaration of input and output streams is explicit in AL and streams may occur at most once on the left-hand-side of an equation. In particular output streams must occur exactly once as a left-hand-side and input streams may not occur as a left-hand-side.

9.9.3 Semantics.

AL is restricted to second-order definitions and has a fixed-point semantics in the style of Kahn.

9.9.4 Language Development and Current Uses.

For an introduction to the use of AL see Section 4.2.2 on the FOCUS project.

A prototype of AL has been implemented on a SUN workstation (see [175]) and experiments to implement AL on an INTEL hyper-cube are in progress (see [83]).

9.9.5 Syntax.

The RS-Flip-Flop can be represented in AL as follows:

```
programme flipflop ≡ chan bool In1, In2 → chan bool Out1, Out2:
    funct nor ≡ bool b1, b2 → bool:
        not(b1 or b2),
    agent streamnor ≡ chan bool sb1, sb2 → chan bool sbout:
        sbout ≡ nor(ft.sb1, ft.sb2)
    end,
    agent leftbs ≡ chan bool lbs1, rbs1 → chan bool lbs
        lbs ≡ lbs1
    end,
    agent rightbs ≡ chan bool lbs1, rbs1 → chan bool rbs
        rbs ≡ rbs1
    end,
    Out1 ≡ true & streamnor(In1, rightbs.flipflop(In1, In2)) & leftbs.rt.flipflop(rt.In1, rt.In2)
    Out2 ≡ false & streamnor(In2, leftbs.flipflop(In1, In2)) & rightbs.rt.flipflop(rt.In1, rt.In2)
end flipflop.
```

9.10 PL

PL is a imperative, parallel procedural language designed for stream programming.

Constructs and Primitives. In some sense PL can be considered to be a classical language containing assignment statements and while loops. However, in addition PL also has the non-terminating loop construct *loop...pool*. PL is syntactically very similar to AL and allows the definition of *functions* and *components* (see Section 9.9) and also has all the stream processing functions described in Section 8.2 as basic operations.

9.10.1 The use of Streams.

As with many stream programming languages variables are used to represent input, but in addition as with AL variables are also used to explicitly represent output. In contrast to AL there are two explicit operators in PL for 'reading' and 'writing' values to and from streams (channels) denoted '?' and '!' respectively that can be

defined informally as follows.

If c is a channel identifier and x is a variable of appropriate type then the command

$$c?x$$

is interpreted informally as 'remove the first value from channel c and assign this value to variable x' If c is empty then execution of this command is delayed (possible infinitely). Similarly if c is again a channel identifier and E is an expression of appropriate type then the command

$$c!E$$

is interpreted informally as 'evaluate E and then write this value to channel c.' Again if E cannot be evaluated, as it may depend on some input evaluation, then this command may also be delayed (possible infinitely).

The use of these two operations provides a model of asynchronous communication and it is pointed out in [45] that they should not be confused with the operators '?' and '!' in *CSP* (see [102]) that provide synchronous communication.

In PL equations are further restricted in that channel identifiers may only occur once (at most) in the right hand side. Also, new channels may be introduced dynamically within PL via recursion and hence dynamic networks may be modelled. For this reason the use of the word channel is less related to the concept of a stream in PL than it is in AL.

9.10.2 Semantics. PL is based on an operational state transformer semantics derived from work in [47] and [62]. It is intended that this semantics can be related to an equivalent abstract (denotational) semantics as a ST and hence PL can be related formally to an AL specification.

9.10.3 Language Development and Current Use. For an introduction to the use of PL see Section 4.2.2 on the FOCUS project.

9.10.4 Syntax. The RS-Flip-Flop can be represented in PL as follows:

```
programme flipflop ≡ chan bool In1, In2 → chan bool Out1, Out2:
    var bool i1, i2, l1, l2;
    var bool o1 := true, o2 := false;
    var nat time := 0;
    loop
        if time > 0 then
            In1?i1;
            In2?i2;
            o1 := i1 nor l2;
            o2 := i2 nor l1;
        fi
        Out1!o1;
        Out2!o2;
        l1 := o1;
        l2 := o2;
        time := time + 1;
    pool
end flip-flop.
```

9.11 Daisy

Daisy (see for example [176], [114], [115], [118], [116] and [113]) is a lazy, higher-order, untyped language based on a 'suspending constructor' (see also [80] and [78]) that has been used extensively for the stream-based programming and synthesis of hardware. In particular, Daisy contains higher-order mapping operators that are optimized for stream filtering, and a demand-driven intermediate constructor that orders lists based on the convergence of suspensions. This latter feature provides a means to express and manage asynchronous concurrency.

9.11.1 Constructs and Primitives. As Daisy is a general purpose functional language it is possible to define all the stream processing functions described in Section 8.2. However, among Daisy's specific primitives are unidirectional device instantiators for terminal screens, keyboards, pipes, sockets and virtual channels to window managers, and (un)scanners and (un)parsers that can be used to iteratively coerce between character streams, symbol streams and expressions.

9.11.2 The Use of Streams. Daisy has a standard functional approach to the definition of first-order streams.

9.11.3 Semantics. As Daisy is a functional language it can be given a standard Kahn style semantics. However, in additional it has a formal calculus for reasoning about and symbolically manipulating Daisy programmes (see for example [232]). Furthermore, the operational interpretation of Daisy programmes may differ from standard functional programmes due to the suspending constructors.

9.11.4 Language Development and Current Uses. Daisy is currently in its third implementation stage (see for example [113]). Most of the current research is concerned with the refinement of the algebra for digital system derivation (see for example [231]). In particular, the development of formal laws for the manipulation of Daisy programmes for hardware synthesis.

9.11.5 Daisy Syntax. The RS-Flip-Flop can be described in Daisy as follows:

```
NOR = /Z. [not *]:[or *]:Z

flipflop = /[S R]. [Qhi Qlo]
    where
        Qhi = [tt ! NOR:[S Qlo]]
        Qlo = [ff ! NOR:[R Qhi]]
```

9.12 PROLOG with streams

[19] and [18] describe a modification of PROLOG (see [136]) (that for convenience we will denote PROLOG) to provide an applicative language for the specification of a class of ADU-SPSs.

9.12.1 Constructs and Primitives. In PROLOG a network of agents is specified by a set of *Horn clauses* wherein each clause corresponds to a particular agent. The structure of

the language is essentially that of PROLOG and the stream processing primitives available are those used in the functional approach (see Section 8.2).

9.12.2 The use of Streams. The approach to streams in PROLOG is the same as that in functional languages. In particular, in PROLOG uni-directional channels are modelled by shared syntactically distinguished input and output variables within each atomic clause and hence the expressive power of PROLOG is limited compared to conventional PROLOG, as invertibility is limited. However, the authors claim that this is not a problem in practice.

9.12.3 Semantics. PROLOG is formalized using a standard fixed-point semantics (see [217]) and makes a explicit distinction between *data constructors* and functions (see [144]) to modify the semantic model to deal with infinite terms.

9.12.4 Language Development and Current Uses. It is intended that PROLOG is viewed as a proper extension of a term re-writing system, wherein each Horn clause is interpreted as an extended re-write rule. It is also the authors' intention that completion algorithms such as Knuth-Bendix (see [128]) can be generalized to generate confluent systems from PROLOG network descriptions. However, we are not aware of any subsequent work by the authors in this field.

9.12.5 Syntax. The RS-Flip-Flop can be represented in PROLOG as follows:

```
type BOOL is tt,ff ;
type STREAM-OF-BOOL is nil, cons(BOOL, STREAM-OF-BOOL) ;

fflop1 : STREAM-OF-BOOL × STREAM-OF-BOOL → STREAM-OF-BOOL ;
fflop2 : STREAM-OF-BOOL × STREAM-OF-BOOL → STREAM-OF-BOOL ;
Nor : BOOL × STREAM-OF-BOOL → BOOL ;
not : BOOL → BOOL ;

    fflop1(cons(b1,sb1),cons(b2,sb2)) = cons(cons(tt,o1),o2) →
        o1 = Nor(b1,fflop2(cons(b1,sb1),cons(b2,sb2))) ;
        o2 = fflop1(sb1,sb2) ;

    fflop2(cons(b1,sb1),cons(b2,sb2)) = cons(cons(ff,o3),o4) →
        o3 = Nor(b2,fflop1(cons(b1,sb1),cons(b2,sb2))) ;
        o4 = fflop2(sb1,sb2) ;

    Nor(ff,cons(b,sb)) = not(b) → ;
    Nor(tt,cons(b,sb)) = ff → ;

    not(tt) = ff → ;
    not(ff) = tt → ;
```

9.13 STREAM

STREAM (see [65] and [66]) is a concurrent scheme language designed for formally specifying, reasoning about and transforming hardware designs at the conceptual, register and gate level. Furthermore, STREAM is intended to address description features associated with each level in a single formalism. The approach is rather like a single programming language that includes formal, high-level and machine-code descriptions as primitives, and is referred to as *almost hierarchical* approach (see [205]).

STREAM is an acronym for STandard REpresentation of Algorithms for Micro-electronics. However, the name STREAM is also intended to reflect the stream processing nature of the language. Indeed, in addition to its role as hardware description language STREAM can also be directly interpreted as a dataflow language, resembling the language of [68]. However, the formal equivalence of STREAM and Dennis's language is not addressed.

9.13.1 Constructs and Primitives. STREAM uses the following stream processing primitives that are referred to as *agents*.

(1) Append. For each $s \in S$ we define the *append agent*

$$\&_s : A_s \times [T \to A_s] \to [T \to A_s]$$

(ambiguously denoted &) by

$$(\forall a \in A_s)\,(\forall a' \in [T \to A_s])\,(\forall t \in T) \quad (a \,\&\, a')(t) = \begin{cases} a & \text{if } t = 0, \text{ and} \\ a'(t-1) & \text{otherwise.} \end{cases}$$

(2) Lifting. For each $\sigma \in \Sigma_{w,s}$ for each $w \in S^+$ and for each $s \in S$ we define the *lifting agent*

$$*_{w,s} \; : (A^w \to A_s) \to ([T \to A^w] \to [T \to A_s])$$

(ambiguously denoted $*$) by

$$(\forall a \in A^w)\,(\forall t \in T) \quad \sigma^*(a)(t) = \sigma(a(t)).$$

(3) Distribution. For each $s \in S$ we define the *distribution agent*

$$\text{distr}_s : [T \to \mathbb{B}] \times [T \to A_s] \rightsquigarrow [T \to A_s] \times [T \to A_s]$$

(ambiguously denoted distr) by

$$(\forall b \in [T \to \mathbb{B}])\,(\forall a \in [T \to A_s])\,(\forall t \in T) \quad \text{distr}(b, a)(t) = (x_1, x_2)$$

wherein

$$x_1 = a(\mu\, k \geq t.[b(k) = tt])$$

and

$$x_2 = a(\mu\, k' \geq t.[b(k') = ff]).$$

(4) **Selection.** For each $s \in S$ we define the *selection agent*

$$\text{selec}_s : [T \to \mathbb{B}] \times [T \to A_s] \to [T \to A_s]$$

(ambiguously denoted selec) by

$$(\forall b \in [T \to \mathbb{B}])(\forall a_1, a_2 \in [T \to A_s])(\forall t \in T) \quad \text{selec}(b, a_1, a_2)(t) = \begin{cases} a_1(t) & \text{if } b(t) = tt, \text{ and} \\ a_2(t) & \text{otherwise.} \end{cases}$$

In addition STREAM also uses the following functional constructs for building SPSs from more primitive SPSs:

(A) **Parallel Composition.** For each $u, u', v, v' \in S^+$ we define the *parallel composition constructor* ambiguously denoted $\Downarrow$ with functionality

$$\Downarrow : ([T \to A^u] \to [T \to A^v]) \times ([T \to A^{u'}] \to [T \to A^{v'}]) \to ([T \to A^{u\,u'}] \to [T \to A^{v\,v'}])$$

by

$$(\forall \mathbb{S} \in [T \to A^u] \to [T \to A^v])\,(\forall \mathbb{S}' \in [T \to A^{u'}] \to [T \to A^{v'}])$$
$$(\forall a = (a_1, \ldots, a_{|u\,u'|}) \in [T \to A^{u\,u'}])\,(\forall t \in T)$$
$$(\mathbb{S} \Downarrow \mathbb{S}')(a)(t) = (x_1, \ldots, x_{|v\,v'|})$$

wherein

$$x_i = \begin{cases} (\mathbb{S}(a_1, \ldots, a_{|u|})(t))_i & \text{if } i \leq |u|, \text{ and} \\ (\mathbb{S}'(a_{|u|+1}, \ldots, a_{|u\,u'|})(t))_i & \text{otherwise.} \end{cases}$$

(B) **Sequential Composition.** For each $u, v, w \in S^+$ we define the *sequential composition constructor* ambiguously denoted $\Rightarrow$ with functionality

$$\Rightarrow : ([T \to A^u] \to [T \to A^v]) \times ([T \to A^v] \to [T \to A^w]) \to ([T \to A^u] \to [T \to A^v])$$

by

$$(\forall \mathbb{S} \in [[T \to A^u] \to [T \to A^v]])\,(\forall \mathbb{S}' \in [[T \to A^v] \to [T \to A^w]])\,(\forall a \in [T \to A^u])\,(\forall t \in T)$$

$$(\mathbb{S} \Rightarrow \mathbb{S}')(a)(t) = \mathbb{S}'(\mathbb{S}(a))(t).$$

(C) **Feedback.** For each $s \in S$ and for each $u, v \in S^+$ we define the *feedback constructor* ambiguously denoted C with functionality

$$\text{C} : ([T \to A^{s\,u}] \to [T \to A^{s\,v}]) \to ([T \to A^u] \to [T \to A^v])$$

by

$$(\forall \mathbb{S} \in [T \to A^{s\,u}] \to [T \to A^{s\,v}])\,(\forall a \in [T \to A^u])\,(\forall t \in T) \quad (\text{C}^{s,u,v}\mathbb{S})(a)(t) = \mathbb{S}(x, a)(t)$$

wherein

$$x = (\mathbb{S}(x, a))_1.$$

Notice here that as x is defined recursively in terms of itself whether $\text{C}(\mathbb{S})$ is computable will depend on the definition of $\mathbb{S}$.

(D) Forking. For each $s \in S$ we define the *fork constructor* ambiguously denoted fork with functionality

$$\mathrm{fork} : [T \to A_s] \to [T \to A_s] \times [T \to A_s]$$

by

$$(\forall a \in [T \to A_s])\,(\forall t \in T) \quad \mathrm{fork}_s(a)(t) = (a(t), a(t)).$$

(E) Permuting. For each $s \in S$ we define the *permutation constructor* ambiguously denoted perm with functionality

$$\mathrm{perm} : [T \to A_s] \times [T \to A_s] \to [T \to A_s] \times [T \to A_s]$$

by

$$(\forall a_1, a_2 \in [T \to A_s])\,(\forall t \in T) \quad \mathrm{perm}_s(a_1, a_2)(t) = (a_2(t), a_1(t)).$$

(F) Sinks. For each $s \in S$ and for each $u \in S^*$ we define the *sink constructor* ambiguously denoted sink with functionality

$$\mathrm{sink} : [T \to A^{s\,u}] \to [T \to A^u]$$

by

$$(\forall a = (a_1, a_2, \ldots, a_{|u|+1}) \in [T \to A^{s\,u}])\,(\forall t \in T) \quad \mathrm{sink}^{s\,u}(a)(t) = (a_2(t), \ldots, a_{|u|+1}(t)).$$

9.13.2 The use of streams. Again in common with the functional approach to stream programing, STREAM adopts the generalized concept of stream as the union of finite and infinite sequences.

9.13.3 Semantics. Both a denotational and algebraic semantics have been derived for STREAM (see [65] and [67] respectively). The denotational semantics is used in [65] to demonstrate the equivalence of STREAM with a procedural language for stream processing.

9.13.4 Language Development and Current Uses. We are not aware of the development of the use of STREAM in hardware design.

9.13.5 Syntax. SIGNAL uses two syntactic styles to reflect the different requirements of hardware description at different levels of abstraction: an applicative style and a functional style. The RS-Flip-Flop can be represented in the two styles as follows:

Applicative.

agent *RS-flipflop* =
  in r, s ***ni***
    $t :\equiv nor^*(r, s'),$
    $r' :\equiv tt \,\&\, t,$
    $u :\equiv nor^*(s, r'),$
    $s' :\equiv \mathit{ff} \,\&\, u$
  out r', s' ***tou***

Functional.

agent *RS-flipflop* =
    $C($
        $C($
            $(perm \Downarrow Id^* \Downarrow Id^*) \Rightarrow$
            $(Id^* \Downarrow perm^* \Downarrow Id^*) \Rightarrow$
            $(nor \Downarrow nor) \Rightarrow$
            $(ff\& \Downarrow tt\&) \Rightarrow$
            $(fork \Downarrow fork) \Rightarrow$
            $(Id^* \Downarrow perm \Downarrow Id^*)$
        $)$
    $)$

9.14 ASTRAL

ASTRAL *A*lgebraic *S*tream *Tra*nsfomer *L*anguage (see [200] – not related to [49]) is intended to provide a formal, equational specification language for STs based on ideas taken from SCA theory (see Section 6.1). In particular, using the theory in [201], ASTRAL has been designed to reconcile the use of AFSTs as a specification technique with the use of CFSTs as a formal semantics (see Section 6.1). This is desirable as by using a CFST semantics in [200] it is shown that the correctness of a broad and non-trivial class of ASTRAL programmes is decidable relative to the use of equational logic augmented with induction and case analysis as a proof system.

Based on these theoretical ideas an implementation of ASTRAL has been developed that in addition to its use for hardware specification is intended to be used as a high-level, declarative, general purpose programming language. (We note that the implementation of ASTRAL actually restricts the user to using primitive recursive definitions to maintain the same theoretical properties as abstract ASTRAL. However, the authors claim [contenciously] that for practical purposes this restriction is not a limitation.) This implementation of ASTRAL is discussed below.

9.14.1 Constructs and Primitives. Because the implementation of ASTRAL is intended to be a general purpose stream programming language it has no language specific stream processing primitives. In contrast, ASTRAL is able to specify (primitive recursive equational forms of) all of the stream processing primitives mentioned in this survey. Indeed, each ASTRAL programme is essentially nothing more than a collection of two types of AFST definition, although non-STs (first-order functions); user-defined data types; and abbreviations to reduce the size and syntactic complexity of programmes are also permitted. As such, we now discuss some of these classes of definitions. However, because ASTRAL's syntax is derived indirectly by compilation into the language *PREQ*, in contrast with our other examples, it is not possible in this survey to indicate the formal semantics of the ASTRAL constructs. The interested reader can consult [200].

Evaluated and Un-Evaluated AFST Definitions, and Function Definitions. All of these classes of definitions have the following basic structure:

Function_name(var_1 : d_type_1,...,var_n : d_type_n) r_type_1,...,r_type_m [(t)]
=

definition_body

wherein '(t)' is only used to indicate an evaluated AFST definition.

Evaluated AFST definitions are the concrete mechanism by which (primitive recursive) AFSTs are specified. A such to insure primitive recursiveness *Function_name* can only be used in the definition body in evaluated form. Sort names in the range of an AFST definition must be stream sorts.

Un-evaluated AFST definitions are the the concrete mechanism by which AFSTs (with a Cartesain form semantics) can be composed. Hence, *Function_name* cannot be used in the function body. Again, sort names in the range of an AFST definition must be stream sorts.

For first-order function definitions this syntax is nothing more than a standard functional specification technique, although the form in which 'function_name' may appear in 'definition_body' is controlled syntactically to preserve primitive recursiveness. Hence, as far as the user is concerned the only syntactic distinction between un-evaluated AFST and function definitions is the range type of the defined function.

There are four basic types of compound expressions that can be used in the body of AFST and function definitions: *case statements*, *ifmatch statement*, *for ... statements* and *for ... while ... statements* that simply restrict the usual general purpose programming primitives to their primitive recursive form. For example, a primitive recursive form of the Lucid primitive *Whenever* can be expressed in ASTRAL as follows:

whenever(s : sortStream, b : boolStream) u_sortStream (t)
=
  s(t) if b(t) = true;
  whenever(s,b)(n) for n = t' to MAX_NAT
    while not b(n);
  u.

wherein 't'' is an abbreviation for '$t+1$', ';' is used as a shorthard for 'or' or 'otherwise' (',' is used as an abbreviation for 'and' – see below); and '*MAX_NAT*' is some pre-defined maximum value that will vary from implementation to implementation.

Pre-Defined and User-Defined Type Definitions. In contrast with most formal algebraic specification languages it is not necessary to make an explicit definition of the underlying signature and variables that are used in an ASTRAL programme. Rather, this information is derived implicitly from each individual ST and function definition. In particular, the standard constants and operations associated with the following pre-defined data types are always available to the user without the need for their explicit inclusion: *bit*, *byte*, *bool*, *char*, *nat* and *int*. In addition, for each of these data types (and also for each user-defined data type), the associated array type, set type, stream type, stream of array type, stream of set type and the data type extended with the undefined element u are also available to the user without their explicit definition. For example: *bitArray*, *bitSet*, *bitStream*, *bitArrayStream*, *u_bitArray*, *u_bitSet*, *u_bitStream* and *u_bitArrayStream* are always available to the user. For technical reasons associated with the automated verification of ASTRAL programmes, the 'real numbers' are supported via a built in library rather than a pre-defined type.

User defined data types come in three basic forms: restrictions of pre-defined types, compound types and type unions.

9.14.2 Language Development and Current Use.

A partial implementation of a prototype ASTRAL specification and verification system is discussed in [200]. Work on the development of a full verification system based on ideas taken from [200] is currently in progress.

9.14.3 ASTRAL Syntax.

Because of the limitiations in describing ASTRAL's constructs concisely, in this section we show how both the Flip-Flop specification and the full Flip-Flop implementation can be specified in ASTRAL including its pre- and post-processing scedules (see Section 9.1). In so doing we note that *RSFlipFlopSpec*, *FFlop*, *OutSch* and *InpSch* are examples of evaluated AFST definitions; and *RSFlipFlopImp* is an example of an un-evaluated AFST definition defined by the composition of other AFSTs.

```
RSFlipFlopSpec(s1,s2 : boolStream) boolStream (t)
=
    true if t = 0;
    false if t > 0, s1(t) = true, s2(t) = false;
    true if t > 0, s1(t) = false, s2(t) = true;
    RSFlipFlopSpec(s1,s1)(t).
```

and

```
RSFlipFlopImp(s1,s2 : boolStream) boolStream boolStream
=
    OutSch(FFlop(true,false,InpSch(s1,s2))).

FFlop(b1,b2 : bool, s1,s2 : boolStream) boolStream boolStream (t)
=
    (b1,b2) if t = 0;
    (FFlop1(b1,b2,s1,s2)(t) nor s2, s1 nor Fflop2(b1,b2,s1,s2)(t)).

OutSch(s1,s2 : boolStream) boolStream (t)
=
    s1(t * 2).

InpSch(s1,s2 : boolStream) boolStream boolStream (t)
=
    (s1(t div 2), s2(t div 2)).
```

10 Conclusions

We have examined the topic of 'stream processing' that we have classified as the study of *stream transformers* STs (second-order functionals) and *stream processing systems* SPSs (implementations of STs typically visualized as directed graphs). We have shown the literature is rich with examples of stream processing research with many different motivating interests, semantic models, and implementation techniques.

However, despite the breadth of existing researach we have shown that in general the literature has concentrated on the examination of the practical properties of SPSs, rather than the theoretical properties of more abstract STs. Therefore, we believe that the literature is still

underdeveloped from certain theoretical perspectives. Indeed, it has been one of our motivations in the writing of this survey to clarify this point, and show that at present no general theory of stream processing exists in an accessible form.

Research that has begun to address more general theoretical considerations is: [4] and [16] that discuss why streams can be a problem for the perspective of foundational mathematics; [213] and [214] that study the generalized computability of stream-based computation; [201] that studies the compositional properties of *Cartesian form* stream transformers; [200] that studies the specification and formal verification of STs; and [160], [161], [162], [163], [164] and [99] that algebraically study the specification and verification of higher-order systems including STs.

References

[1] H Abelson and G Sussman. *The Structure and Analysis of Computer Programs.* MIT Press, 1985.

[2] S Abramsky, editor. *Reasoning about concurrent systems.* 1983.

[3] W B Ackerman. Data flow languages. In R E Merwin, editor, *AFIPS National Computer Conference*, volume 48, pages 1087–1095, 1979.

[4] P Aczel. *Non-well-founded Sets.* CLSI Lecture Notes. University of Chicago Press, 1988.

[5] D Adams. *A computation model with data flow sequencing.* PhD thesis, Stanford University, December 1969.

[6] D Adams. A model for parallel computations. In L C Hobbs *et al*, editor, *Parallel Processor Systems, Technologies, and Applications*, pages 311–333. Spartan, 1970.

[7] M Amamiya, R Hasegawa, and S Ono. VALID: A High Level Functional Language for Dataflow Machines. *Rev ECL*, 32(5):793–802, 1984.

[8] P Amblard and H Charles. Music Synthesis Description with the Data Flow Language LUSTRE. *Microprocessing and Microprogramming*, 27:551–556, 1989.

[9] Arvind and K P Gostelow. Some Relationships between Asychronous Interpreters of a Dataflow Language. In E J Neuhold, editor, *Formal Description of Programming Concepts*, pages 95–119. North-Holland, 1978.

[10] Arvind, K P Gostelow, and W Plouffe. An asynchronous programming language and computing machine. Department of Information and Computer Science Technical Report 114A, University of California, 1979.

[11] E A Ashcroft, A A Faustini, R Jagannathan, and W W Wadge. *Multidimensional Programming.* Oxford University Press, 1995.

[12] J Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, August 1978.

[13] J Backus. Is computer science based on the wrong fundamental concept of programme? In J W de Bakker and J C van Vilet, editors, *Algorithmic languages*, pages 133–165. Elsevier North-Holland, Amsterdam, 1981.

[14] M Bartha. An Algebraic Model of Synchronous Systems. *Information and Computation*, 97:97–131, 1992.

[15] M Bartha. Foundations of a theory of synchronous systems. *Theoretical Computer Science*, 100:325–346, 1992.

[16] J Barwise and L Moss. Hypersets. *The Mathematical Inteligencer*, 13(4):31–41, 1991.

[17] R A Becker and J M Chambers. *S: An Interactive Environment for Data Analysis and Graphics.* Wadsworth Inc, Belmont CA, 1984.

[18] M Bellia, E Dameri, P Degano, G Levi, and M Martelli. A Formal Model For Lazy Implementations of a Prolog-Compatible Functional Language. In J A Campbell, editor, *Implementations of PROLOG*, Artificial Intelligence, pages 309–340. Ellis Harwood, 1984.

[19] M Bellia, E Dameri, and M Martelli. Applicative Communicating Processes In First Order Logic. In G Goos and J Hartmanis, editors, *International Symposium on Programming*, volume 137, pages 1–14. Springer-Verlag, 1982.

[20] M Bellia and G Levi. The Relation Between Logic and Functional Languages: A Survey. *Journal of Logic Programming*, 6(3):217–236, October 1986.

[21] J Bergstra and J W Klop. Process Algebra for the Operational Semantics of Static Data Flow Networks. Mathematical Centre Technical Report IW 222/83, University of Amsterdam, 1983.

[22] J A Bergstra and J V Tucker. Equational Specifications, complete term rewriting systems, and computable and semicomputable algebras. University College of Swansea, Computer Science Division, Technical Report CSR 20–92, University College of Swansea, 1992.

[23] G Berry. A hardware implementation of pure Esterel. In *International Workshop on Formal Methods in VLSI*, Lecture Notes In Computer Science, 1991.

[24] G Berry and L Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In S D Brookes A W Roscoe and G Winkel, editors, *Seminar on Concurrency*, number 197 in LNCS, pages 389–448. Springer-Verlag, 1984.

[25] G Berry, P Couronne, and G Gonthier. *Synchronous Programming of Reactive Systems: an Introduction to ESTEREL*, pages 35–56. Elsevier Science Publishers BV (North Holland), 1988.

[26] G Berry and G Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *The Science of Computer Programming*, 842, 1988.

[27] G Berry and G Gonthier. Incremental Development of an HDLC entity in Esterel. *Computer Networks*, 22:35–49, 1991.

[28] A Beveniste and G Berry. The Synchronous approach to reactive and real-time systems. *Proceedings IEEE*, 79:1270–1282, 1991.

[29] J Blom. Tools for the Specification and Analysis of Coupled Map Lattices. Final Year Dissertation, Department of Computer Science, University College of Swansea, 1992.

[30] J Blom, A V Holden, M J Poole, J V Tucker, and H Zhang. Caress II: a general purpose tool for parallel deterministic systems, with applications to simulating cellular systems. *Journal of Physiology (London)*, 467:145, 1993. Leeds Meeting 14-15 Jan.

[31] G Boudol, V Roy, R de Simone, and D Vergamini. Process calculi, from theory to practice: verification tools. In *Automatic verification methods for finite state systems*, number 407 in Lecture Notes In Computer Science, pages 1–10. Springer-Verlag, 1990.

[32] F Boussinot and R de Simone. The Esterel language. In *Another look at real time languages*, number 79 in Proceedings of the IEEE, pages 1293–1304, 1991.

[33] J D Brock. Operational Semantics of a Data Flow Language. MIT Lab for Computer Science Technical Memo 120, MIT, 1987.

[34] J D Brock and W B Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of Programming Concepts*, number 107 in Lecture Notes in Computer Science. Springer-Verlag, 1981.

[35] M Broy. Applicative real time programming. In *Information Processing 83, IFIP World Congress, Paris*, pages 259–264. North Holland, 1983.

[36] M Broy. A theory for nondeterminism, parallelism, communication and concurrency. *Theoretical Computer Science*, pages 1–61, 1986.

[37] M Broy. Predicative specification for functional programs describing communicating networks. *Information Processing Letters*, 25:93–101, 1987.

[38] M Broy. Semantics of finite or infinite networks of communicating agents. *Distributed Computing*, 2:13–31, 1987.

[39] M Broy. An example for the design of a distributed system in a formal setting - the lift problem. Technical Report MIP 8802, Universität Passau, February 1988.

[40] M Broy. Non-deterministic data flow programs: How to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.

[41] M Broy. Towards a Design Methodology for Distributed System. In M Broy, editor, *Constructive Methods In Computer Science*, volume 55 of *NATO ASI Series F: Computer and System Sciences*, pages 311–364. Springer-Verlag, 1989.

[42] M Broy. Functional Specification of Time Sensitive Communicating Systems. In G Rozenburg, J W de Bakker, and W P de Roever, editors, *Stepwise Refinement of Distributed Systems*, volume 420 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[43] M Broy. Compositional refinement of interactive systems. Working Material for the International Summer School on Program Design Calculi, 1992.

[44] M Broy. (Inter-)action refinement. Working material for the International Summer School on Program Design Calculi, 1992.

[45] M Broy, F Dederichs, C Dendorfer, M Fuchs, F Gritzer, and W Weber. The Design of Distributed Systems: An Introduction to FOCUS. Technical Report TUM-19292-2, Technische Universität Müchen, January 1993.

[46] M Broy and C Dendorfer. Modelling operating system structures by timed stream processing functions. *Journal of Functional Programming*, 2(1):1–21, January 1992.

[47] M Broy and C Lengauer. On Denotational verses Predicative Semantics. *Journal of Computer and System Sciences*, 42(1):1–29, 1991.

[48] R E Bryant. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computers*, C-33(2):160–177, 1984.

[49] G Buonanno, A Coen-Porisini, and W Fornaciari. Hardware specification using the assertion language ASTRAL. In P Prinetto and P Camurati, editors, *Correct Hardware Design Methodologies*, pages 335–358. North-Holland, 1992.

[50] W H Burge. Stream Processing Functions. *IBM Journal of Research and Development*, pages 12–25, 1975.

[51] R M Burstall, J S Collins, and R J Popplestone. *Programming in POP-2*. Edinburgh University Press, 1971.

[52] R M Burstall, D B MacQueen, and D T Sanella. HOPE: an Experimental Applicative Language. In *Lisp Conference, Stanford*, 1980.

[53] P Caspi, D Pilaud, N Halbwachs, and J A Plaice. LUSTRE: A Declarative Language for Programming Synchronous Systems. 14th *ACM Symposium on Principles of Programming Languages, Munich*, pages 178–188, January 1987.

[54] K Clark and S Gregory. Notes on the Implementation of PARLOG. *Journal of Logic Programming*, 2(1):17–42, 1985.

[55] K L Clark and S Gregory. PARLOG: A Parallel Logic Programming Language. Research Report 83/5, Imperial College, May 1983.

[56] K L Clark and S A Gregory. A Relational Language for Parallel Programming. In *ACM Conference on Functional Programming and Computer Architecture*, pages 171–178, 1981.

[57] D Clement and J Incerpi. Programming the behaviour of graphical objects using Esterel. In *Proceedings TAPSOFT*, number 352 in Lecture Notes In Computer Science. Springer-Verlag, 1989.

[58] A Cohn and M Gordon. A Mechanized Proof of Correctness of a Simple Counter. In K McEvoy and J V Tucker, editors, *Theoretical Foundations for VLSI Design*, volume 10 of *Tract in Theoretical Computer Science*, pages 65–96. Cambridge University Press, 1990.

[59] N J Cutland. *Computability : an Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[60] R B Dannenberg. Artic: A Functional Language for Real-Time Control. In *Symposium on Lisp and Functional Programming*, 1984.

[61] W P de Roever. On backtracking and greatest fixed points. In E J Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 621–639. North-Holland, 1978.

[62] F Dederichs. Transformation verteiler Systeme: Von applikativen zu prozeduralen Darstellungen. Technical Report SFB 342/17/92, Technische Universität München, 1992.

[63] D DeGroot and G Lindstrom. *Logic Programming: Functions, Equations, and Relations*. Prentice Hall, 1986.

[64] C Delgado Kloos. *Towards a Formalism of Digital Circuit Design*. PhD thesis, Technische Universität München, 1986.

[65] C Delgado Kloos. Semantics of Digital Circuits. In G Goos and J Hartmanis, editors, *Semantics of Digital Circuits*, volume 285 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[66] C Delgado Kloos. STREAM: A Scheme Language for Formally Describing Digital Circuits. In G Goos and J Hartmanis, editors, *PARLE Parallel Architecture and Languages Europe*, volume 2 of *Lecture Notes in Computer Science*, pages 333–350. Springer-Verlag, 1987.

[67] C Delgado Kloos, W Dosch, and B Möller. On the Algebraic Specification of a Language for Describing Communicating Agents. In *ÖGI/ÖCG Conference, Passau*, pages 53–73, February 1986.

[68] J B Dennis. First Version of a Data Flow Procedure Language. In B Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1974.

[69] J Derrick, G Lajos, and J V Tucker. Specification and verification of synchronous concurrent algorithms using the Nuprl proof development system. Centre for theoretical computer science report, University of Leeds, 1989.

[70] C Dezan, H Le Verge, P Quinton, and Y Saouter. The Alpha du Centaur Experiment. In P Quinton and Y Robert, editors, *Algorithms and Parallel VLSI Architectures II*, pages 325–334. Elsevier Science Publishers, 1992.

[71] P Dyber. Program verification in a logical theory of constructions. In J Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in LNCS, pages 334–349. Springer-Verlag, 1985.

[72] P Dyber and H Sander. A functional programming approach to the specification and verification of concurrent systems. Technical report, Chalmers University of Technology and University of Göteborg, Department of Computer Sciences, 1988.

[73] S M Eker, V Stavridou, and J V Tucker. Verification of synchronous concurrent algorithms using OBJ3. A case study of the Pixel Planes architecture. In *Proceedings of Workshop on Designing Correct Circuits*, Oxford, 1990. Springer-Verlag.

[74] S M Eker and J V Tucker. Specification and verification of synchronous concurrent algorithms: a case study of the Pixel Planes architecture. In R A Earnshaw P M Dew and T R Heywood, editors, *Parallel processing for computer vision and display*, pages 16–49. Addison Wesley, 1989.

[75] M Farah. *Correct Compilation of a Useful Subset of Lucid.* PhD thesis, University of Waterloo, Ontario, Canada, 1977.

[76] A Faustini. *The Equivalence of a denotational and an operational semantics for pure dataflow.* PhD thesis, University of Warwick, Computer Science Department, Coventry, United Kingdom, 1982.

[77] R B France. Semantically Extended Data Flow Diagrams - A Formal Specification. *IEEE Transactions on Software Engineering*, 18(4):329–346, 1992.

[78] D P Friedman and D S Wise. Cons should not evaluate its arguments. In *Proceedings of the 3rd Colloquium on Automata, Languages and Programming*, pages 257–284. Edinburgh Press, 1976.

[79] D P Friedman and D S Wise. Aspects of applicative programming for file systems. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 41–55. ACM SIGPLAN, 1977.

[80] D P Friedman and D S Wise. Unbounded computational structures. *Software-Practice and Experience*, 8(4):407–416, 1977.

[81] J A Goguen. OBJ as a Theorem Prover with Applications to Hardware Verification. In 2*nd Banff Workshop on Hardware Verification*, Banff Canada, June 1987.

[82] M Gordon, R Milner, and C Wadsworth. Edinburgh LCF. In *Semantics of Concurrent Computation*, number 70 in LNCS. Springer-Verlag, 1979.

[83] S Gorlatch. Parallel Program Development for a recursive numerical algorithm: a case study. Technical Report SFB 342/7/92, Technische Universität München, March 1992.

[84] T Guatier, P Le Guernia, and L Besnard. Signal: A Declarative Language For Synchronous Programming Of Real-Time Systems. Technical report, IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE, 1987.

[85] P Le Guernic, T Gautier, M LeBorgne, and C LeMaire. Programming real time applications with Signal. *Proceedings IEEE*, pages 1321–1336, 1991.

[86] J R Gurd, J R W Glauert, and C C Kirkham. Generation of dataflow graphical object code for the Lapse programming language. In W Händler, editor, *CONPAR 81*, number 111 in Lecture Notes In Computer Science, pages 155–168. Springer-Verlag, June 1981.

[87] H Halbwachs, F Lagnier, and C Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, 1992.

[88] N Halbwachs, P Caspi, P Raymond, and D Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings IEEE*, 79:1305–1320, September 1991.

[89] K Hanna and N Daeche. Strongly-Typed Theory of Structures and Behaviours. In G Milne and L Pierre, editors, *Correct Hardware Design and Verification Methods*, Lecture Notes In Computer Science, pages 39–54. Springer-Verlag, 1993.

[90] C H Hansson. Case studies in the specification and correctness of neural networks. Final year dissertation, University College Swansea, 1993.

[91] D Harel and A Pnueli. On the Development of Reactive Systems. Weizmann Institute of Science, Rehovot, Israel, 1985.

[92] N A Harman. *Formal Specifications for Digital Systems*. PhD thesis, School of Computer Studies, University of Leeds, 1989.

[93] N A Harman and J V Tucker. In *Formal Specification and the Design of Verifiable Computers*, Proceedings of the 1988 UK IT Conference, pages 500–503, University College of Swansea, 1988. IEE.

[94] N A Harman and J V Tucker. Clocks, Retimings, and the Formal Specification of a UART. In G J Milne, editor, *The Fusion of Hardware Design and Verification*. North-Holland, 1988.

[95] N A Harman and J V Tucker. Formal specifications and the design of verifiable computers. In *Proceedings of 1988 UK IT Conference*, pages 500–503. Institute of Electrical Engineers (IEE), 1988. Held under the auspices of the Information Engineering Directorate of the Department of Trade and Industry (DTI).

[96] N A Harman and J V Tucker. The Formal Specification of a Digital Correlator I: Abstract User Specification. In K McEvoy and J V Tucker, editors, *Theoretical Foundations for VLSI Design*, volume 10 of *Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[97] N A Harman and J V Tucker. Consistent Refinements of Specifications for Digital Systems. In P Prinetto and P Camurati, editors, *Correct Hardware Design Methodologies*, pages 273–295. North-Holland, 1992.

[98] D Harrison. RUTH: A Functional Language For Real-Time Programming. In G Goos and J Hartmanis, editors, *PARLE Parallel Architectures and Languages Europe*, number 259 in LNCS, pages 297–314. Springer-Verlag, 1987.

[99] B M Hearn and K Meinke. ATLAS: A Typed Language for Algebraic Specification. In J Heering K Meinke, B Möller, and T Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting: First International Workshop, HOA '93, Amsterdam, The Netherlands*, Lecture Notes in Computer Science, 816, pages 146–168. Springer Verlag, Berlin, 1994.

[100] P Henderson and J H Morris. A lazy evaluator. In 3rd *Conference on the Principles of Programming Languages*, pages 95–103. ACM, 1976.

[101] J Herath, N Saito, K Toda, Y Yamaguchi, and T Yuba. Data-flow computing base language with n-value logic. In *Fall Joint Comp Conf*, 1986.

[102] C A R Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[103] K M Hobley. *The Specification and Verification of Synchronous Concurrent Algorithms*. PhD thesis, School of Computer Studies, University of Leeds, 1990.

[104] K M Hobley, B C Thompson, and J V Tucker. Specification and verification of synchronous concurrent algorithms: a case study of a convolution algorithm. In G Milne, editor, *The fusion of hardware design and verification*, pages 347–374. North-Holland, 1988. Proceedings of IFIP Working Group 10.2 Working Conference.

[105] M Holcombe. X-Machines as a Basis for System Specification. *Software Engineering*, 3(2):69–76, 1988.

[106] M Holcombe and F Ipate. X-machines with stacks and recursive enumerable functions. Technical report, Department of Computer Science, University of Sheffield, England, 1994.

[107] A V Holden, M J Poole, J V Tucker, and H Zhang. Coupling CMLs and the Synchronization of a Multilayer Neural Computing System. *Chaos, Solitons and Fractals*, 1993. To appear.

[108] A V Holden, J V Tucker, and B C Thompson. Can excitable media be considered as computational systems? *Physica D*, 49:240–246, 1991.

[109] A V Holden, J V Tucker, and B C Thompson. The computational structure of neural systems. In A V Holden and V I Kryukov, editors, *Neurocomputers and Attention I: Neurobiology, Synchronisation and Chaos*, pages 223–240. Manchester University Press, 1991.

[110] A V Holden, J V Tucker, H Zhang, and M J Poole. Coupled Map Lattices as Computational Systems. *Chaos*, 2:367–376, 1992.

[111] AV Holden, B C Thompson, J V Tucker, D Withington, and H Zhang. A Theoretical Framework for Synchronization, Coherence and Chaos in Real and Simulated Neural Networks. In J Taylor, editor, *Workshop on Complex Dynamics in Neural Networks*, pages 223–240. Springer-Verlag, 1992.

[112] R Jagannathan. *A Descriptive and Prescriptive Model for Dataflow Semantics.* PhD thesis, Department of Computer Science, University of Waterloo, 1988.

[113] E Jeschke. *An Architecture for Parallel Symbolic Processing based on Suspending Construction.* PhD thesis, Department of Computer Science, Indiana University, 1995.

[114] S D Johnson. *Synthesis of Digital Designs from Recursion Equations.* MIT Press, 1983.

[115] S D Johnson. *Synthesis of Digital Designs from Recursion Equations.* MIT Press, 1984.

[116] S D Johnson. Manipulating Logical Organization with System Factorizations. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*. Springer-Verlag, 1990.

[117] S D Johnson and Z Zhu. An Algebraic Approach to Hardware Specification and Derivation. In L Claeson, editor, *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*. Elsevier, 1991.

[118] C B Jones. *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science. Prentice Hall, 1986.

[119] S B Jones and A F Sinclair. Functional programming and Operating Systems. *The Computer Journal*, 32, 1989.

[120] B Jonsson. A fully abstract trace model for dataflow networks. Technical Report 88016, Swedish Institute of Computer Science, 1988.

[121] G Kahn. The semantics of a simple language for parallel processing. *Proceedings IFIP Congress*, pages 471–475, 1974.

[122] G Kahn. A Primitive for the Control of Logic Programs. In *Proceedings of the Symposium on Logic Programing*, pages 242–251, Atlantic City, 1984. IEEE Computer Society.

[123] G Kahn and D B MacQueen. Coroutines and networks of parallel processes. IFIP Congress, pages 993–998. Elsevier North-Holland, Amsterdam, 1977.

[124] Y Kamp and M Hasler. *Recursive Neural Networks for Associative Memory.* Wiley, 1990.

[125] P Kearney and J Staples. An Extensional Fixed-Point Semantics For Non-Deterministic Data Flow. *Theoretical Computer Science*, 91(2):129–179, 1991.

[126] R M Keller. *Denotational models for parallel programs with ideterminate operators*, pages 249–312. North Holland, 1978.

[127] C Kiliminster. Lecture: History of Computation Colloquium, Oxford University, 1993.

[128] D Knuth and P Bendix. Simple Word Problems in Universal Algebra. In J Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[129] T Kohonen. Correlation matrix memories. In J A Anderson and E Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 174–180. MIT Press, 1972.

[130] T Kohonen. *Associative Memory: A System Theoretical Approach.* Springer Verlag, 1978.

[131] J N Kok. A Fully Abstract Semantics for Data Flow Nets. In G Goos and J Hartmanis, editors, *PARLE Parallel Languages and Architectures Europe*, volume 2, pages 351–368. Springer-Verlag, 1987.

[132] J N Kok. A Fully Abstract Semantics for Data Flow Nets. In G Goos and J Hartmanis, editors, *PARLE Parallel Languages and Architectures Europe*, volume 2, pages 351–368. Springer-Verlag, 1987.

[133] W H F J Körver. *A Discrete Switch-Level Circuit Model that uses 4-valued node states.* PhD thesis, Technische Universiteit Eindhoven, 1993.

[134] W H F J Körver. A Discrete Formalization of Switch-Level Circuit Behavior. *Integration, the VLSI Journal*, 1995. (to appear).

[135] P R Kosiniski. A data flow programming language for operating systems. *Proceedings ACM, Sigplan-Sigops Interface Meeting, Sigplan Notices*, 8(9):89–94, September 1973.

[136] R A Kowalski. *Predicate Logic as a Programming Language*, pages 569–574. North-Holland, 1974.

[137] H T Kung. Why systolic architectures. *Computer*, pages 37–46, January 1982.

[138] P J Landin. The Correspondence Between ALGOL 60 and Church's Lambda Calculus: Part 2. *Communications of the ACM*, 8:158–165, 1965.

[139] P J Landin. The Correspondence Between ALGOL 60 and Church's Lambda Notation: Part 1. *Communications of the ACM*, 8:89–101, 1965.

[140] P J Landin. The next 700 programming languages. *Communications of the Association for Computing Machinery*, 9:157–166, 1966.

[141] P T Lee and K P Tan. Modeling of Visualized Data-Flow Diagrams Using Petri Net Model. *Software Engineering*, 7(1):4–12, 1992.

[142] C E Leiserson and J B Saxe. Optimizing Synchronous Systems. *VLSI Computing Systems*, 1:41–67, 1983.

[143] G Levi and C Palamidessi. Contributions to the Semantics of Logic Perpetual Processes. *Acta Informatica*, 25:691–711, 1988.

[144] G Levi and A Pegna. Top-Down Mathematical Semantics and Symbolic Execution. *RAIRO Inform. Théor*, 17:55–70, 1983.

[145] P-Y P Li and A J Martin. The Sync Model: A parallel Execution Method for Logic Programming. In *Symposium on Logic Programming*, pages 223–234, Salt Lake City, 1986. IEEE Computer Society.

[146] G Lindstrom and P Panangaden. Stream-Based Execution of Logic Programe. In *Proceedings of the Symposium on Logic Programming*, pages 168–176, Atlantic City, 1984. IEEE Computer Society.

[147] J W Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[148] Z Manna, S Ness, and J Vuillemin. Inductive methods for proving properties of programs. *Communications of the Association for Computing Machinery*, 16(8):491–502, August 1973.

[149] D Marshall. CADISP: Cellular Automata Design Implementation and Specification. Final Year Dissertation, Department of Computer Science, University College of Swansea, 1991.

[150] A R Martin. *The Specification and Simulation of Synchronous Concurrent Algorithms.* PhD thesis, School of Computer Studies, University of Leeds, 1989.

[151] A R Martin and J V Tucker. The concurrent assignment representation of synchronous systems. *Parallel Computing*, 9:227–256, 1988.

[152] B McConnell. *Infinite Synchronous Concurrent Algorithms.* PhD thesis, Computer Science Department, University College of Swansea, 1993.

[153] B McConnell and J V Tucker. Infinite Synchronous Concurrent Algorithms: The Algebraic Specification and Verification of a Hardware Stack. In F L Bauer, W Brauer, and H Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 321–375. Springer-Verlag, 1993.

[154] W S McCulloch and W Pitts. A logical calculus of the ideas immanent in nervous activity. In J A Anderson and E Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 18–27. MIT Press, 1943.

[155] K McEvoy and J V Tucker. Theoretical foundations of hardware design. In K McEvoy and J V Tucker, editors, *Theoretical foundations of VLSI design.* Cambridge University Press, 1990.

[156] J McGraw, S Skedzielewski, S Allan, R Oldhoeft, J Glauert, C C Kirkham, B Noyce, and R Thomas. SISAL: Streams and Iteration in a Single Assignment Language. Language Reference Manual, Version 1.2, Lawrence Livermore National Laboratory, California, March 1985.

[157] M C McIlroy. 'Coroutines'. Internal report, Bell Telephone Laboratories, Murray Hill, New Jersey, 1968.

[158] C Mead and J Conway. *Introduction to VLSI systems.* Addison-Wesley, 1980.

[159] K Meinke. *A Graph Theoretic Model of Synchronous Concurrent Algorithms.* PhD thesis, School of Computer Studies, University of Leeds, 1988.

[160] K Meinke. Equational Specification of Abstract Types and Combinators. In G Jäger, editor, *Computer Science Logic '91*, Lecture Notes in Computer Science, 626. Springer Verlag, Berlin, 1991.

[161] K Meinke. Algebraic semantics of rewriting terms and types. In M Rusinowitch and J-L Remy, editors, *Third International Conference on Conditional Term Rewriting Systems*, Lecture Notes in Computer Science, 656. Springer Verlag, Berlin, 1992.

[162] K Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100:385–417, 1992.

[163] K Meinke. A second order initial algebra specification of primitive recursion. *Acta Informatica*, 31:329–340, 1994.

[164] K Meinke and L J Steggles. Specification and Verification in Higher Order Algebra: A Case Study of Convolution. In J Heering, K Meinke, B Möller, and T Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting: First International Workshop, HOA '93, Amsterdam, The Netherlands*, Lecture Notes in Computer Science, 816, pages 189–122. Springer Verlag, Berlin, 1994.

[165] K Meinke and J V Tucker. Specification and representation of synchronous concurrent algorithms. In F H Vogt, editor, *Concurrency '88*, number 335 in Lecture Notes in Computer Science, pages 163–180. Springer-Verlag, 1988.

[166] G K Milne. Timing Constraints: Formalizing their Description and Verification. In *Proceedings of Computer Hardware Description Languages and their Applications.* North-Holland, 1989.

[167] R Milner. Model of LCF. Technical report, Computer Science Department, Stanford University, 1973.

[168] R Milner. A Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

[169] R Milner. A Proposal for Standard ML. In *ACM Symposium on LISP and Functional Programming*, pages 184–197, Austin, Texas, 1984.

[170] R Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[171] M Minsky and S Papert. *Perceptrons.* MIT Press, 1969.

[172] D P Misunas. Deadlock Avoidance in Data Flow Architecture. In *Proc. Symp. Automat. Computation and Contr.*, pages 337–343, April 1975.

[173] G Murakami and R Sethi. Terminal call processing in Esterel. Technical Report 150, AT and T Bell Laboratories, 1990.

[174] L Naish. All Solutions Predicates in Prolog. In *Proceedings of the Symposium on Logic Programming*, pages 73–77, Boston, 1985. IEEE Computer Society.

[175] H Nueckel. Eine Zeigerimplementierung von Graphreduktion für eine Datenflußprache. Diploma thesis, Universität Passau, 1988.

[176] M J O'Donnell. Circuits and Systems: Implementing Communication with Streams. In M Ruschitzka, editor, *IMACS Transactions on Scientific Computation*, volume 2, pages 311–319. 1983.

[177] M J O'Donnell. Hydra: Hardware description in a functional language using recursion equations and high order combining forms. In G J Milne, editor, *The Fusion of Hardware Design and Verification*. North-Holland, 1988.

[178] A C Parker. Automated Synthesis of Digital Systems. *IEEE Design and Test of Computers*, pages 75–81, November 1984.

[179] D S Parker. *Stream Data Ananlysis in Prolog*, chapter 8, pages 249–312. MIT Press, 1990.

[180] G D Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Århus University, Århus, Denmark, September 1981.

[181] A Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J W de Bakker, W P de Roever, and G Rozenberg, editors, *Current Trends in Concurrency*, number 224 in Lecture Notes in computer Science, pages 510–584. Springer-Verlag, 1986.

[182] M J Poole. *Synchronous Concurrect Algorithms and Dynamical Systems*. PhD thesis, Computer Science Department, University College of Swansea, October 1994.

[183] A Rabinovich. On the Schematological Equivalence of Dataflow Networks. In *Computer Science and Logic*, 1993.

[184] C Ratel, N Halbwachs, and P Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language lustre. In *Conference Software for Critical Systems*, ACM-SIGSOFT, 1991.

[185] D E Rumelhart, G E Hinton, and R J Williams. Learning internal representation by error propagation. In D E Rumelhart and J L McClelland, editors, *Parallel Distributed Processing. Explorations in the Microstructure of Cognition. Volume 1: Foundations*, pages 318–362. MIT Press, 1986.

[186] D E Rumelhart and J L McClelland, editors. *Parallel Distributed Processing. Explorations in the Microstructure of Cognition. Volume 2: Psychological and Biological Models*. MIT Press, 1986.

[187] D E Rumelhart and J L McClelland, editors. *Parallel Distributed Processing. Explorations in the Microstructure of Cognition. Volume 1: Foundations*. MIT Press, 1986.

[188] J Sargeant. Implementation of Structured Lucid on a Dataflow Computer. Master's thesis, University of Manchester, 1982.

[189] E Y Shapiro. A Subset of Concurrent Prolog and it Interpretor. Technical Report TR-003, ICOT, 1983.

[190] J A Sharp, editor. *Data Flow Computing.* Ablex Publishing Corporation, 1991.

[191] R Sharp and O Rasmussen. Transformational rewriting with ruby. *IFIP Transactions A-Computer Science and Technology*, 32:243–260, 1993.

[192] M Sheeran. *μFP, an Algebraic VLSI Design Language.* D. phil., St. Cross College, November 1983.

[193] M Sheeran. RUBY - a Language of Relations and Higher Order Functions. Technical report, Glasgow University, 1986.

[194] M Sheeran. Retiming and slowdown in Ruby. In G J Milne, editor, *The Fusion of Hardware Design and Verification.* North-Holland, 1988.

[195] M Sheeran. Categories for the working hardware designer. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*. Springer-Verlag, 1990.

[196] F S K Silbermann and B Jayaraman. A domain-theoretic approach to functional and logic programming. *Journal of Functional Programming*, 2(3):273–321, July 1992.

[197] J Staples and V L Nguyen. A Fixed-point Semantics For Nondeterministic Data Flow. *Journal For The Association For Computing Machinery*, 32(2):411–444, 1985.

[198] G Stefanescu. On Flowchart Theories: Part I. The Deterministic Case. *Journal of Computer and System Sciences*, 35:163–191, 1987.

[199] G Stefanescu. On Flowchart Theories: Part II. The Nondeterministic Case. *Theoretical Computer Science*, 52:307–340, 1987.

[200] R Stephens. *Algebraic Stream Processing.* PhD thesis, University College of Swansea, 1994.

[201] R Stephens and B C Thompson. Cartesian Stream Transformer Composition. University College of Swansea, Department of Computer Science Report CSR 21-92, University College of Swansea, 1992. To appear in *Fundamenta Informaticae.*

[202] V Stoltenberg-Hansen, E Griffor, and I Lindeström. *Mathematical Theory of Domains.* Cambridge Tracts in Theoretical Computer Science, 1994.

[203] P A Subrahmanyam and J H You. FUNLOG = Functions + Logic: A Computational Model Integrating Functional and Logic Programming. In *International Symposium on Logic Programming*, pages 144–153. IEEE Computer Soc. Press, 1984.

[204] Sun Microsystems Incorporated. *YACC*, 1988.

[205] G J Sussman and G L Steele. CONSTRAINTS - A Language for Expressing Almost Hierarchical Descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[206] B C Thompson. *A Mathematical Theory of Synchronous Concurrent Algorithms*. PhD thesis, School of Computer Studies, University of Leeds, 1987.

[207] B C Thompson and J V Tucker. Theoretical Considerations in Algorithm Design. In R.A.Earnshaw, editor, *NATO ASI Fundamental Algorithms for Computer Graphics*. Springer-Verlag, 1985.

[208] B C Thompson and J V Tucker. Equational specification of synchronous concurrent algorithms and architectures. Computer Science Division, Technical Report CSR 9.91, University College of Swansea, 1991.

[209] B C Thompson and J V Tucker. Equational specifications of synchonous concurrent algorithms and architectures (second edition). Technical Report CSR 9.91, Computer Science Department, University of Wales, Swansea, 1994.

[210] B C Thompson, J V Tucker, and W B Yates. Artificial Neural Networks as Synchronous Concurrent Algorithms: Algebraic Specification of a FeedForward Backpropogation Network. Technical report, University College of Swansea Computer Science Division Research Report, 1992.

[211] C M N Tofts. The Relationship Between Synchronous Concurrent Algorithms and SCCS. Technical report, Department of Computer Science, University College of Swansea, 1993.

[212] J V Tucker and J I Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*. North Holland, 1988.

[213] J V Tucker and J I Zucker. Theory of Computation over Stream Algebras and its Applications. In I M Havel and V Koubek, editors, *Mathematical Foundations of Computer Science 1992: 17th International Symposium, Prague*, LNCS, pages 62–80. Springer-Verlag, 1992.

[214] J V Tucker and J I Zucker. Computable Functions on Stream Algebras. In H Schwichtenburg, editor, *International Summer School on Proof and Computation, Marktoberdorf, 1993*, NATO Advanced Study Institute, pages 341–382. Springer-Verlag, 1994.

[215] D A Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Language and Computer Architetcures*, Lecture Notes in Computer Science, 201. Springer-Verlag, 1985.

[216] M H van Emden and G T de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In K L Clark and S A Tarnlund, editors, *Logic Programming*, pages 189–198. Academic Press, 1982.

[217] M H van Emden and R A Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23:733–742, 1976.

[218] H Le Verge, C Mauras, and P Quinton. The ALPHA Language and its Use for the Design of Systolic Arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.

[219] J von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[220] J Vuillemin. *Proof techniques for recursive programs.* PhD thesis, Stanford University, 1973.

[221] W Wadge. Viscid, a vi-like Screen Editor written in pLucid. Technical Report DCS-40-IR, University of Victoria, Computer Science Department, 1984.

[222] W W Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13:3–15, 1981.

[223] W W Wadge and E A Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.

[224] W P Weijland. Verification of a Systolic Algorithm in Process Algebra. In K McEvoy and J V Tucker, editors, *Theoretical Foundations of VLSI Design*, volume 10 of *Tracts in Theoretical Computer Science.* Cambridge University Press, 1990.

[225] P H Welch. Parallel Assignment Revisited. *Software - Practice and Experience*, 13:1175–1180, 1983.

[226] K S Weng. Stream orientated computation in recursive data flow schemas. Project MAC Technical Memo 69, MIT, 1975.

[227] B Widrow and M E Hoff. Adaptive switching circuits. In J A Anderson and E Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 126–134. MIT Press, 1960.

[228] D Winkel and F Prosser. *The Art of Digital Design.* Prentice-Hall, 1987.

[229] S Wolfram. *Theory and applications of cellular automata.* World Scientific, 1986.

[230] W B Yates. *Algebraic Specification and Correctness of Artificial Neural Networks and Supervised Learning Algorithms.* PhD thesis, University College of Swansea, 1993.

[231] Z Zhu and S D Johnson. An Algebraic Framework for Data Abstraction in Hardware Description. In Sheeran and Jones, editors, *Proceedings of the Oxford Workshop on Designing Correct Circuits.* Springer-Verlag, 1990.

[232] Z Zhu and S D Johnson. An Example of Interactive Hardware Transformation. In Subramanyam, editor, *Proceedings of ACM International Workshop on Formal Methods in VLSI Design*, 1991.