FORMAL VERIFICATION SUPPORT FOR ELLA

IED 4/1/1357

Semantics and Verification Foundations for ELLA[1]

University of Manchester[2] RSRE

13th May 1991

[2]The University of Manchester acknowledges support from the SERC via research grant GR/F 38174.

Contents

Part I

Introduction

Chapter 1

ELLA and hardware verification

This chapter provides an introductory overview of work performed so far under tracks 1 and 2 of the project. The organisation of this deliverable is also presented.

ELLA is a system pioneered by RSRE for the high-level description of complex VLSI designs [MPT85a]. The system[Com90a], which is commercially available[1] includes the ELLA language and its compiler, an application support environment, a simulator and other design support tools. ELLA is a parallel language and is used to describe and model circuit behaviour by defining the nodes, connections and signal flows in a circuit network.

The capabilities of modern silicon processing have increased in recent years to the extent that it is now very difficult - if not impossible - to understand the functionality and design of VLSI digital circuits when they are described at the gate level. Hardware Description Languages (HDLs) such as ELLA have allowed the level of abstraction at which circuits are described to be raised and enabled designers to cope with these increasing levels of complexity. Design systems like ELLA provide a common environment in which designs can be refined and simulated from high levels of abstraction down to the gate level.

The use of HDLs alone will not be enough in the future to exploit fully the opportunities offered by VLSI. Designers need to be relieved of some of the more mundane design tasks to enable them to concentrate on the system or higher level aspects of design. This is the area in which synthesis and testability assessment tools have much to offer. Increasing design automation does not however solve the problem of checking the correctness of designs.

The problem of design verification, whether checking a complete chip implementation against a specification or checking the correctness of a synthesised logic design, can only be tackled through the use of mathematically sound proof techniques. Attempts to reason about designs described in HDLs have been hindered to date by treating the language semantics on an informal basis.

The work we present here lays part of the foundation of a larger collaborative project, namely, to develop a design support environment including design tools based on formal verification techniques, capable of handling ELLA designs of up to 100K gates. Clearly this size presents a major challenge to the current verification technology. Sophisticated techniques need to be applied in order to extend the capabilities of the particular approach towards the desired goal. Those aspects, however, will be the subject of later reports and papers. Here we wish to concentrate on introducing the basic ideas.

Hardware verification will be used here to describe the process of determining whether a physical computing device (such as a VLSI chip) exhibits the behaviour required by the device User. This implies that hardware verification will be involved at every stage in the design process. It is to be distinguished

[1] Originally marketed by Praxis Systems, more recently through Computer General Electronic Design Ltd (CGED)

from what shall be called *formal verification*. This is the process of determining that an abstract mathematical model representing, in this case, a computing device possesses a set of properties, and whether two models are equivalent in certain well-defined ways. Clearly, formal verification will form a major part of hardware verification.

Methods for hardware verification will range from animation — supplying an executable representation of the design with a finite set of inputs and analysing the output produced — to using tools for performing theorem-proving or equivalence checking applied to formal models of the design. A design system should therefore provide an integrated simulation and formal verification toolset. In current design systems, designers are limited to using animation only for predicting the behaviour of a design.

[SBE90] and [Gog88] provide useful commentaries on the features which should be provided by a verification system. The following list describes the useful properties of specification languages and verification methods:

- integration of verification and specification techniques into existing VLSI CAD tools
- simplicity and expressive power of specification language, leading to natural hardware descriptions (which will be easier for the non-expert to follow)
- hierarchical specification structures, allowing (de)composition
- executable specifications needed; this allows the designer or user to exercise the specification, and to gain confidence in it
- traceback facility during theorem proving, so that the designer can determine where either the design violated the theorem, or where the theorem prover failed
- concise proof traces

ELLA is, syntactically, a large language and comprises many different language features. However, many of these features can be reformulated in terms of "more primitive" features. Naturally, one of the first steps towards our project goal has been to identify the basic language elements to which we can expect to apply formal verification. This has led to Core ELLA [HM91] and more recently to Kernel ELLA. However, before one can contemplate formal verification of program properties, a formal semantics for the programming language is a necessity. In the case of ELLA, as with other HDLs, this does not currently exist. There is obviously some, good, informal understanding of desired semantics (in plural), and indeed, there is a *de facto* standard implied by the CGED ELLA simulator[2]. However, this does not assist with formal verification at all.

Recently there have been three independent approaches to providing a formal semantics for ELLA. At Edinburgh operational semantics are provided for two simplified versions of ELLA, called *micro*ELLA[Goo90a] and *pico*ELLA[Goo90b]. At Cambridge, a system is being developed to translate ELLA-text into HOL[BGHT90], so that the HOL verification system and its associated tools can be employed. At Manchester, we gave a structured operational semantics (SOS) for a small parallel language with delays[BGM91]. That work was designed to capture the fundamental semantic aspects of ELLA and stood as a pilot scheme towards providing a full reference semantics for Core ELLA. The principal purpose of the formal semantics, for our project at least, is not to act as a reference in that sense; it is, rather, to act as a semantic foundation for verification.

[2] To some extent, it is fortunate there is only one marketed simulator available.

1.1 Organisation of Volume

We have structured this volume into four parts each containing several chapters, according to the main areas of work conducted in the project so far.

Part I is concerned with the background to the project and contains this introductory chapter followed by chapter 2 which examines current industrial hardware design processes that are based on ELLA. In particular, it establishes a set of designer requirements and a set of verification requirements together with a discussion of methods needed to fulfil them.

Part II gives the language definitions for Core ELLA, Kernel ELLA, and Boolean Kernel ELLA (*BKE*). Chapter 3 provides a definition of Core ELLA, a subset of ELLA containing primitive orthogonal constructs. All complex ELLA constructs can still be transformed into Core ELLA, although it is small enough to facilitate formal verification. This chapter also outlines the transformation process. Core ELLA must however still address the problems of scope and type-checking. These are removed in Chapter 4, where a set of data structures is defined to be Kernel ELLA. A set of transformations between Core ELLA and Kernel is then given. Chapter 5 introduces *BKE*, providing a composite syntax for it. *BKE*is an ELLA-like language containing recursive lets, delays, function declarations and case statements. The purpose of *BKE* is to act as a foundation of formal verification of ELLA descriptions using state-based techniques.

Part III considers the development of two formal semantics for *BKE*, and therefore for ELLA. Chapter 6 provides an IO automaton semantics for *BKE*. The chapter also includes IO automata representations of simple logic circuits, and shows how these are combined using an automaton product operation. In contrast, Chapter 7 describes a history-based semantics for *BKE*. This is an operational semantics constructed using the SOS notation, developed initially to act as a reference semantics for ELLA and to provide the basis for a reference simulator. A necessary subject for future study is to compare these two approaches, and ultimately prove their equivalence. This process is outlined in Chapter 8. *BKE* at present does not contain all of the language features of Kernel ELLA. The main omission is a full typing mechanism and related operations. Chapter 9 outlines how it is proposed to include these in future versions of *BKE*, so that *BKE*will correspond exactly with Kernel ELLA.

Part IV focuses on formal verification, using *BKE* and the automaton-based semantics. Chapter 10 describes and illustrates the technique for ELLA-to-ELLA verification, so that two different ELLA representations of a design can be checked for equivalence. Having transformed an ELLA description into *BKE* and then into an IO automaton, standard algorithms are available to determine automaton equivalence or containment. An SR latch and a twisted ring counter are used to illustrate the process. Two descriptions for each are considered and then their respective IO automata are shown to be equivalent. The chapter also describes how the automata for these are constructed from their components. Chapter 11 outlines future work on ELLA-to-ELLA transformation, as a basis for verification. Chapter 12 shows formal verification of simple temporal and ordering properties, using the twisted ring counter as an example. Chapter 13 and Chapter 14 outline areas of future study covering different aspects of the verification process. Chapter 13 considers methods for generating test vectors from semantics, and using IO automaton semantics as a simulator generator. Chapter 14 considers abstraction, hiding and composition during design. Finally Chapter 15 considers representation techniques necessary to help reduce complexity of IO automaton based verification. This represents the start of the work required to produce a system capable of handling large ELLA designs.

A collection of Appendices contains useful reference material to accompany the main text. This includes a summary of verification requirements, the composite syntax for ELLA and Core ELLA, and the set of data structures for Kernel ELLA. Two example applications are given of ELLA transformations, outlined in Chapter 3, and transforming ELLA descriptions into Core ELLA.

1.1.1 Warning

We have assumed throughout this volume that the reader is reasonably familiar with the various topics including the HDDL ELLA, propositional calculus and first order predicate calculus, basic temporal logic and some elementary automata theory. For the reader who wishes to recap on these topics we suggest, respectively, the following references [Com90b], [Sch67], [Gab91] and [HU79a].

Chapter 2

Verification: requirements and techniques

This chapter introduces the ELLA hardware description language and discusses basic issues in the verification of hardware designs. It examines current industrial hardware design processes based upon the ELLA hardware description language and proposes a set of Designer Requirements for future design tools and support, studying the role that ELLA plays. The nature of hardware verification, involving the use of both informal and formal methods, is examined. A set of Verification Requirements was also formed and is included as Appendix A. Finally, the verification approach adopted in this project is briefly described and motivated.

2.1 Introduction

What is required of a design environment which caters for *hardware verification*? How would a design system carry out the tasks required? What would be the best way for a user to interact with the system? This chapter examines these issues, providing background information, and forms a set of Verification Requirements. Possible solutions are suggested, including those to be used in this project.

The work is based on the results from three meetings with industrial design engineers, where ELLA has been employed as part of each design process. The first meeting was held at Plessey Research, Caswell, with two subsequent meetings at Racal Research, Reading, and STL, Harlow. The aim of the meetings was to study the hardware design processes currently used, to determine the role that ELLA plays, and to establish a set of Designer Requirements for future design support. Further details of the three meetings can be found in [Wil90].

It should be noted at the outset that all of the sites were involved in designing devices with digital signal-processing (DSP) applications, and that this is likely to influence approaches to design, and requirements of the design systems. Other external factors which may affect a design process in a commercial environment include constraints such as delivery dates and cost.

2.2 Industrial Design Process

In this section, a general industrial design process will be described, based on results from the visits to Plessey, Racal and STL. The design processes all make use of ELLA, whose role will be examined.

It is possible to use ELLA throughout most of the design process, from capture of the initial user specification, down to design representation at the gate-level. It is found to be particularly useful for design partitioning, even for multi-chip systems, and for exploring architectural issues. At gate-level the design is converted into a format suitable for input to traditional CAD tools, such as standard cell layout. For

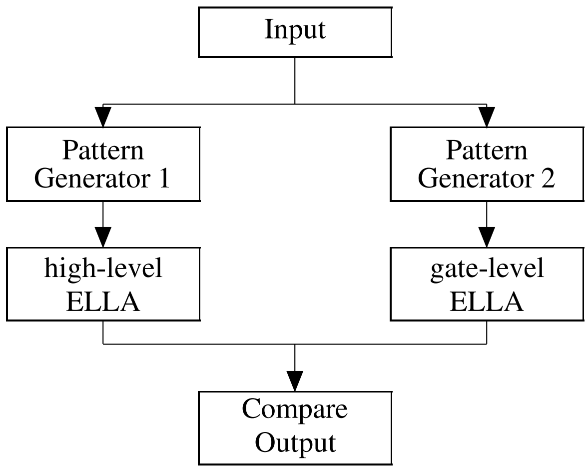

Figure 2.1: Design Verification by Parallel Simulation

example, at Plessey, a design is constructed in parallel using the semi-custom sea-of-gates Plessey Design System (PDS). At the other sites, behavioural models are constructed in ELLA for standard cells of particular CAD tool libraries (for example, VTI, AT&T, Silvar-Lisco, LSI Logic), to facilitate translation of the ELLA design.

The basic design process can be summarised as follows:

1. Specification of design using ELLA or high-level programming language, such as Pascal or C.

2. Partition design into manageable blocks with well-defined interfaces.

3. Produce and verify high-level ELLA model

4. Produce and verify intermediate-level ELLA model

5. Produce and verify hardware-level model

6. Layout

Translation between different design representations is performed using ad hoc translation programs, or possibly by using synthesis tools such as LOCAM, in the case of ELLA-to-ELLA translations. A problem thus arises when changes are made to the new design format, because either the initial ELLA representation must be updated so that it is equivalent to the new representation, or it is obsolete and should be discarded. A design hierarchy is therefore not often maintained, because updates to previously produced designs would have to be made manually, and are thus too time-consuming and error-prone to make them worthwhile. Therefore, the current design representation is often the only version that is retained.

Equivalence between different representations is checked for by comparing simulation results. Figure 2.1 shows a typical test harness for checking the equivalence of two designs. The degree of difficulty in carrying out such a test depends on the similarity in format of the input and output of each design representation. Problems are caused by retiming and rescheduling of data, or by changes in datatypes. The use of abstraction in design is not facilitated by these subsequent problems in design checking and verification.

The initial ELLA description, together with a carefully selected set of test vectors, may form the basis of a contract between them and the user. If the user has access to an ELLA system then the ELLA description can be embedded into a system-level description. Otherwise the ELLA-text forms a concise textual description. However, at STL for example, the initial design descriptions are written in a standard high level programming language, such as Pascal, because the customer is far more likely to have access to the programming language than to the ELLA system.

The ELLA simulator is used for high-level examination and exploration of the design, because it is time-independent and thus allows the designer to concentrate on architectural issues. When real-time analysis is required then a specialised simulator or timing analyser is used. The time-independence of the ELLA simulator is thus seen to be at once an advantage, when considering architecture for example, and a disadvantage, clearly when circuit delay estimates are required. However, even when time-independent simulation is needed, the ELLA simulator was sometimes criticised for being too slow. For example, at STL more use is made of high level programming languages for simulation at different design stages, because they are able to run faster than the ELLA simulator. However, this involves additional effort to ensure that equivalence is maintained between the different design representations.

A collection of file management systems, such as the VMS Code Management System used at Plessey, and ad hoc programs are used to control source code dependencies during a design. A house style is also adopted so that separate parts of a design can be joined together more easily (for example, using common formats for ELLA-text files). These mechanisms all assist in maintaining code-compatibility in a multiple designer system throughout the design process.

Figure 2.2 shows an overview of the use of different design tools during the design processes of these industrial sites. High level programming representations $P_0 \ldots P_k$ are optionally used for design entry or high level simulation. The ELLA-text representations $E_0 \ldots E_m$ are used during design exploration, experiment and translation. CAD tool representations $C_0 \ldots C_n$ are used during low-level simulation, timing analysis and of course layout.

2.2.1 Designer Requirements

From the preceding description of current industrial design processes, it can be observed that the main problems in design arise because there is a shortage of tools and techniques to support design translation, equivalence checking, abstraction and timing. This deficiency leads to the widespread use of animation and simulation in order to check for the correctness of a design as it progresses from the initial user requirements through to silicon.

The first requirement is for tools to support reliable design translation. In current systems, translations occur between two ELLA design representations, two non-ELLA representations, a non-ELLA and an ELLA representation, or between a low-level ELLA and a CAD tool representation. Tools are needed both to perform translations effectively and to check that the resulting design is equivalent in some desired respect to the original design. They should support both data and timing abstraction.

A test pattern comparison tool is required to support parallel simulation in the test harnesses currently used. The tool should cater for retiming and abstraction. Although the emphasis currently placed upon simulation will be reduced when the above tools for design translation and equivalence checking are introduced, simulation will still have an important role to play in the design process, for example during user requirements capture. Any future design representations used should therefore be executable and equipped with the necessary simulation tools.

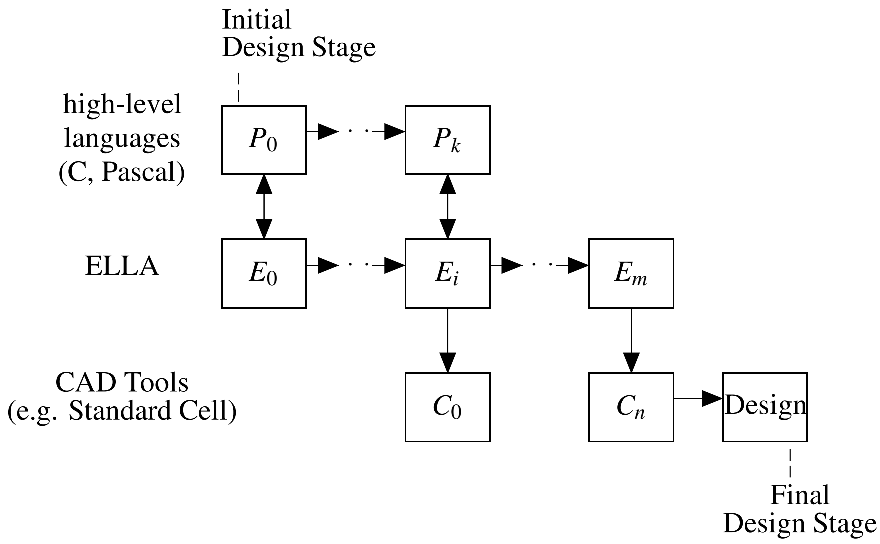

Figure 2.2: General Design Process

A design management system is required so that all aspects during a design cycle can be controlled. It should allow the house design style and procedure to be defined. The system will be able to ensure design integrity, performing history version control and handling multi-user designs.

Some form of schematic capture, especially at high-level, may be of assistance in design, but only if it is flexible enough to support the often informal nature of design exploration. It must therefore cater for partially complete and incorrect designs during their development.

A summary of Designer Requirements appears in the appendix.

2.2.2 Verification Requirements

Three main stages may be identified in the translation of a design from initial user requirements to silicon:

Stage A: User Requirements to User Specification (Requirement Capture)

Stage B: Transformations applied to abstract design representation; for example:

(i) Specification S_1 to Specification S_2

(ii) Specification S to Draft Design D

(iii) Draft Design D_1 to Draft Design D_2

Stage C: Implementable Design D to Physical Device

Hardware verification affects all stages of the design process. Each stage requires different techniques to assist in design translation and to help ensure that the translation preserves correctness. Formal verification methods are only suitable for application during Stage B of design translation, when designs are represented using abstract models. For verification during design stages A and C it is necessary to use other methods, such as animation.

During User Requirements Capture (design Stage A) every effort must be made to facilitate user understanding of the initial user specification. It is therefore necessary to use a simple specification language, which gives a 'natural' description of hardware, from the User viewpoint. The language will be executable so that the initial design description can be animated. It will support abstraction so that unnecessary details, such as real-time issues, can be excluded from the specification. A multi-media system, which allows textual, pictorial and schematic elements to be associated with particular parts of a design, would be useful; user requirements may well be presented initially in such a form. User-controllable presentations of animation results may be of assistance, whilst machine-processable formats would enable that results from design animation to be embedded into the controlling environment.

The core of any future design system supporting hardware verification will be a formal hardware specification and description language, with possibly more than one language being used. This will permit mathematical methods to be used for formal verification, abstraction and design animation, during Stage B. Again it should be expressive and easy for a hardware designer to use.

Another major part of verification will be through the use of animation As stated above this will require tools for test pattern generation retiming or rescheduling, and data abstraction. In conjunction with this, a well-defined design process is necessary so that the status of each design component, such as the location of the current version, its verification status or update history, can be monitored.

At the physical implementation Stage C, there is currently the most support available from traditional CAD tools. However it is not possible to relate the frequently informal and empirically-based models used here, with the low-level discrete models used during design translation in Stage B. Development is required of low-level discrete design language which can be embedded in Stage B, whilst producing animation results in agreement with physical device behaviour, perhaps at specific read-points. Experimentation will be needed to establish simulation accuracy and monitor this as new fabrication techniques evolve.

Post-fabrication testing methods are required to identify devices containing fabrication flaws. These will include a methodology for test pattern generation to ensure that all of the device is exercised and that there is maximum fault coverage.

A lot of effort is currently spent on recording design dependencies and modifications. It is essential that a complete design is constructed from the intended versions of its components, and that the design project library is not altered until the required version of a component is fully tested and approved. The correctness of a design therefore depends on the integrity of its design management system. It is necessary to impose a suitable structure onto a design history, so that the relationship between different design parts and versions, together with any design decisions taken in making a transition, are recorded.

A summary of Verification Requirements is given in the appendix.

2.3 Formal Verification Methods

In this section a selection of formal verification methods, currently being developed for application to hardware design, is examined.

2.3.1 Temporal Logic

Temporal logic is an extension of predicate calculus which includes temporal logical operators, such as *Always* ($\Box$) or *Sometime* ($\Diamond$). A collection of states shows the assignment of values to logical variables at each given time-step. Temporal logical formulae determine how these values of logical variables will change through time, and thus how the states are related. Temporal logic is therefore well-suited for expressing finite state properties of hardware. Different varieties of temporal logic have been studied, for example linear-time or branching time. For linear-time temporal logic there is only one successor state for each state, whilst in branching time temporal logic each state may have more than one successor state.

Methods for verification based on temporal logic, include model-checking, are algorithmic and therefore machine-processable requiring no intervention. They also provide traceback or counter-example facilities which are available to guide the designer if a proof fails, showing an example of failure. There are several systems based on this approach, which vary according to the underlying temporal logic used. This method is used in this project and is described more fully in the next section. The system is ideally suited to proving design properties such as safety and liveness, which declare what a design must never do and what it must eventually achieve respectively. However the main problem with this approach is that of state explosion, leading to intractably large solution sizes. This is an area being addressed in current research, investigating ways for reducing the number of states required in the state-machine representation of a design or property. For example, in [HK90] it is proposed that *homomorphic reductions* are used to cut down the number of states needed each time in the state-machine being checked. An example is given where the n-state input of a machine whose operation only depends on the parity of the input can be reduced to two states by a suitable mapping which depends only on the parity of the input value. The aim of the work is support the specification and verification of communication protocols.

The system is not however well-suited to analysing purely functional or combinatorial design properties which do not have any associated state.

Another related area of research concerns using temporal logic as a language for programming and simulation. These languages are designed so that they can be executed in a similar manner to any usual programming language. It is then possible to simulate designs. In [Mos85] the language *Tempura* is described which is based on interval temporal logic. At Manchester, the METATEM system has been developed[BFG+89a] which is based on propositional linear temporal logic.

2.3.2 Equational Reasoning

OBJ[GW88, Gog88] is a declarative high level programming and specification language based on abstract data types and equational logic, together with an operational semantics based on term-rewriting. This leads to a simple and natural description of hardware. Proofs for equations describing design properties can be constructed using tactics such as term-rewriting and induction. This is a semi-automatic process, however, requiring user guidance and therefore a knowledge of the proof process. The proof construction may well be problematic or lead to very long proof traces. This has the advantage over fully-automatic theorem provers that the user *is* able to guide the development of a proof, rather than having to rely on the inbuilt rules of the particular automatic theorem prover used, which may have difficulty in constructing the proof.

OBJ is being used as the basis of a research project at Royal Holloway and Bedford New College, which aims to develop techniques for the formal verification of designs described in hardware description languages such as ELLA.

2.3.3 Higher Order Logic

HOL is a proof generating system which uses a higher order logic for the expression of design properties; this is then interfaced to the ML programming language, in which proof strategies are encoded[CGM86]. Designs are expressed in higher-order logic, as predicates with function arguments. HOL then utilises standard logical laws and tools of mathematics to generate proofs, constructed using natural deduction. The system has existed for a number of years and so it has already been equipped with a collection of useful pre-proved facts. As with the OBJ system, however, user-intervention may be needed to guide the development of a proof, and therefore the user must have adequate knowledge of the system. Proof traces may again be long.

The HOL system is well-developed and has been used in realistic design case-studies involving formal verification. For example in [Coh87] it is used to verify the equivalence of different abstract models of the Viper microprocessor developed at RSRE[Ker84]. At Cambridge, two research projects, [TH89] and [BGHT90], are currently investigating HOL as a means of providing formal semantics for the hardware description languages VHDL and ELLA. The latter uses the HOL facility of *semantic embedding* to provide the semantics.

2.4 Verification Approach used in this Project

The principal verification approach to be used in this project is one based on model checking. Figure 2.3 presents the general architecture of the approach.

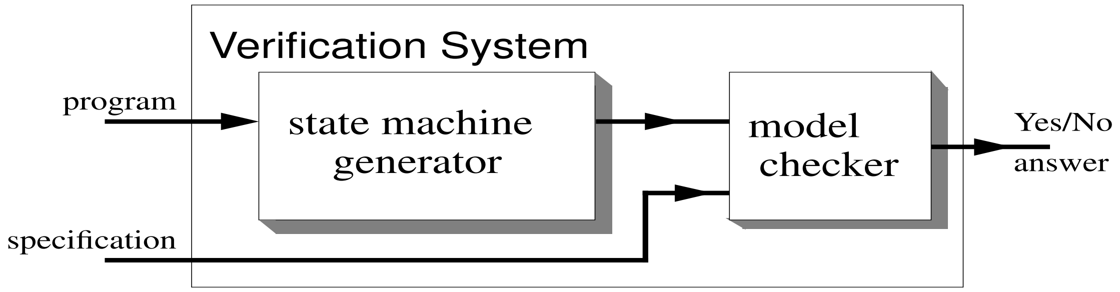

Figure 2.3: Verification System Architecture

The state machine generator takes as input a program and generates a finite state graph which represents all possible runs of the program. Such state graphs can, of course, only be generated for finite state programs i.e. those with bounded memory requirements. This graph is one of the inputs to the model checker, the other being a specification, or property, of the program. The task of the model checker is to determine whether the state graph, and hence the program, possesses the property. There are many approaches to this task, varying according to the characteristics of the specification language. We outline one that fits well with a range of specification formalisms. The idea is that we first construct, from the given specification, a state graph that characterises all runs of programs that *do not* possess the given property. This task is reasonably straightforward, although it can be computationally expensive. We then determine whether the intersection of the program state graph and the latter graph is empty, i.e. whether there are any runs common to both. If the intersection is empty then we can conclude that *all* runs of the program possess the desired property. If the intersection is non-empty then we have a run which does not possess the property, which can be returned to the user as a counterexample. This form of architecture is illustrated in Figure 2.4.

There are several existing systems based on this approach. Clarke et al [CE82, CES83, BC86] developed model checkers for a branching time logic, CTL, for verifying properties of programs written in their state machine language, SML. Sifakis et al [RRSV87] have developed Xésar, a model checking

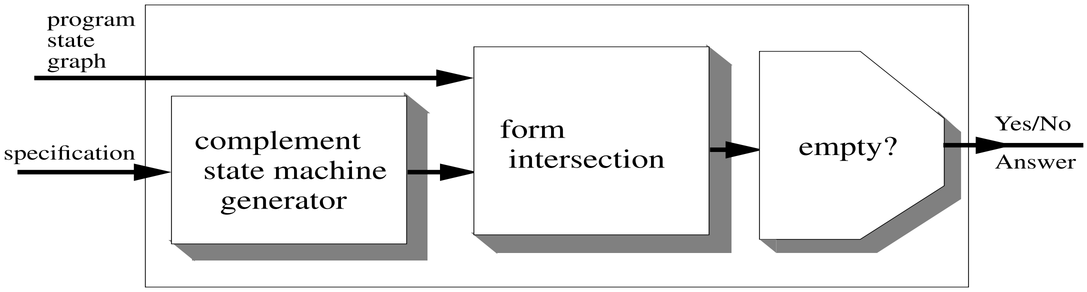

Figure 2.4: Simplified Model Checker Architecture

based verification system for the language ESTELLE, a specification language for describing communications protocols. In Manchester, we have developed a generic system SMG [GB88, BFG89b] which allows the user to specify the input language syntax and semantics, and provides model checking support for linear time temporal logics. This generic approach was not used for ELLA, since the compilation process would not have been efficient enough for use with large designs.

In the above architecture, the specification is usually given in a language, such as temporal logic, suitable for describing particular desired aspects of the programs behaviour. This language could, however, be the original programming language itself, in which case the success of the emptiness test described above would mean that all runs of the program were also runs of the "specification" program. This corresponds to one notion of program refinement. However, in the particular case of ELLA, a more direct and efficient approach may be taken to refinement testing, since all ELLA programs are deterministic.

An automata-theoretic approach is used because, as indicated above, it provides automated verification procedures and has certainly been shown to be successful in the area of communication protocols. It will easily lend itself to verifying equivalence and refinement, in appropriate senses, of ELLA descriptions, as well as enabling verification against formal specification written in more expressive formalisms, such as temporal logics. Furthermore, there is a natural correspondence between automata and ELLA components, which is exploited in the formal semantic definitions appearing in the rest of this paper. A first step in this direction is contained in [Bec87].

2.4.1 Limitations of Hardware Verification

By fulfilling the hardware verification requirements and introducing the verification methods described in this chapter, then the likelihood that a fabricated device will meet the initial requirements of its user should be greatly increased. However, hardware verification can only be performed informally overall, for it is not possible to give a precise meaning to the user requirements, and so it is impossible to reason about them formally (see [Coh89] for a discussion). At the implementation stage, it is also not possible to precisely model the physical behaviour of the VLSI device. Even if it were, then the unavoidable introduction of fabrication flaws would prevent exact prediction of chip behaviour. According to [Gog88], formal verification can also only ever hope to *reduce* the chances of certain errors being introduced during the translation of the initial abstract model of user requirements to the final abstract model of the behaviour of the physical device.

Part II

Language Definitions

Chapter 3

Core ELLA

This chapter introduces Core ELLA, the subset of ELLA which will form the basis of the verification work. Core ELLA is that subset of the full language for which virtually all ELLA descriptions may be transformed into, using existing software tools. The composite syntax for Core ELLA and a brief discussion of well formedness of expressions in the Core is presented

3.1 Introduction

In this part of the deliverable we present the definitions of the data structure and languages which will be used in the development of the verification system. Their definitions are derived from the complete ELLA language the composite syntax of which is given in appendix B. For further information on the complete language the reader is referred to [Com90a]. In this chapter the definition is given of Core ELLA. Core ELLA is a subset of the complete language which is small enough for formal definition and manipulation, yet large enough to retain the character and flavour of the complete ELLA language. In essence Core ELLA is that part of the ELLA language which is arrived at from any ELLA description by applying a set of software transformations. This is possible since ELLA's evolution has, to a large extent, occurred by defining the semantics of the more sophisticated syntactic constructs in terms of the semantics of more primitive constructs. ELLA has thus been able to retain a small number of primitive orthogonal constructs as its nucleus, with the complex constructs transforming onto this nucleus. Thus Core ELLA, though a much simpler language than ELLA, contains the essense of the complete language.

Although Core ELLA is a simpler language than the full ELLA language it still needs to address the problems of scopes and type checking. To this end a set of data structures have been defined which removes such obstacles. This is acheived by introducing an environment which provides every identifier and each type with a unique number. Thus scopes and types are effectively flatterend so that the **Kernel** can identify anything by its unique representation. The set of data structures is defined to be the **Kernel** of ELLA and it is possible to map any ELLA description down onto structures in the **Kernel**. The definition of the **Kernel** along with the formal transformations to go from Core ELLA to the **Kernel** are given in the next chapter.

Since the **Kernel** is a set of data structures all the information it holds cannot easily be expressed in terms of a concrete syntax. However it is possible to consider subsets of the **Kernel** for which a simple syntax is possible. In the last chapter of this part of the deliverable a language using only a boolean type, which will be known as the Boolean Kernel of ELLA, or BKE will be described. This language forms the stepping stone to the **Kernel** and the remaining automaton results of this deliverable.

In the next section of this chapter a brief outline of the ELLA transformational system is given. The following sections then subdivide the ELLA language into different classes to facilitate the definition of

Core ELLA.

3.2 ELLA Transformational System

The ELLA Transformational System is a collection of software procedures which take high level language constructs and transform them into lower level constructs. The transformed circuit description can then be viewed by printing the result in ELLA textual form. This Transformational System has allowed ELLA to incorporate into the language a number of sophisticated syntactic constructs by simply relating their semantic interpretation to already existing constructs. The present set of transforms can be broadly categorised into the following, **macro**, **sequence**, **step**, **function-type**, **timescale** and **imports**.

The **macro** transformation replaces calls of macros with the appropriate function instance with parameters substituted. It also transforms out replicators and evaluates constant and integer expressions. Although it is possible to separate the macro and replicator transforms into two distinct transforms they have been kept together because replicators are a particular form of macro. The **sequence** transformation replaces all the sequence constructs by appropriate lets, makes, joins and unit delays. The **step** transformation replaces all the multiple lets, makes and complex join statements by occurrences of their simpler versions. The **function-type** transformation removes all occurrences of function type signals, for example a single bidirectional wire is replaced by two distinct wires. Use of **timescale** transforms the hierarchical retiming constructs (i.e. FASTER and SLOWER) into functions with delays and a sample-and-hold construct, whilst **imports** substitutes the bodies of imported functions into their appropriate calling function.

As a simple example of the transformational system consider the following description of an RS flip-flop

$$Q_{t+1} = \neg \text{ Reset} \wedge (\text{Set} \vee Q_t)$$
$$\bar{Q}_{t+1} = \neg \; (\text{Set} \vee Q_t)$$

which can be described using ELLA's sequential format in the following way

```
TYPE bool = NEW ( t | f ).

FN RSFF = (bool: set reset) -> [2]bool:
( SEQ
   PVAR q ::= ?bool, qbar ::= ?bool;  # create/initialise state variables  #

   CASE (set, q)                          # Qbar = NOT( Set OR Q )  #
   OF   (f, f)                : qbar := t,
        (t, t)|(f, t)|(t, f)  : qbar := f
   ELSE qbar := ?bool
   ESAC;

   CASE (set, reset)              #   Q =  NOT(Reset) AND ( Set OR Q )    #
   OF   (f, t) : q := f,
        (t, f) : q := t,
        (f, f) :                  # state variable retains previous value #
   ELSE q := ?bool
   ESAC;
```

```
   OUTPUT (q, qbar)               # output new value of state variables   #
).
```

when this is passed through the transformations the sequence constructs are replaced by equivalent parallel constructs to produce the following circuit

```
FN _DELAY = ( bool ) -> bool: DELAY( ?bool, 1 ).

FN RSFF = ( bool: set reset ) -> [ 2 ]bool:
(
    MAKE _DELAY : _s4q.
    MAKE _DELAY : _s5qbar.
    LET qbar = CASE ( set, _s4q )
                 OF ( f, f ):
                    ( LET _s7qbar = t.
                      OUTPUT _s7qbar
                    ),
                    ( t, t ) | ( f, t ) | ( t, f ):
                    ( LET _s8qbar = f.
                      OUTPUT _s8qbar
                    )
               ELSE (
                   LET _s9qbar = ?bool.
                   OUTPUT _s9qbar
                    )
               ESAC.
    LET q = CASE ( set, reset )
              OF ( f, t ):
                 ( LET _s11q = f.
                   OUTPUT _s11q
                 ),
                 ( t, f ):
                 ( LET _s12q = t.
                   OUTPUT _s12q
                 ),
                 ( f, f ): _s4q
            ELSE (
                LET _s13q = ?bool.
                OUTPUT _s13q
                 )
            ESAC.
    JOIN qbar -> _s5qbar.
    JOIN q -> _s4q.
    OUTPUT ( q, qbar )
).
```

where names with a '_' in front of them are generated by the transformation process. It should be noted that the code generated has been designed for simulation purposes rather than readability.

The amount of code generated by the transformations depends on how concisely the higher level description has been written. For example the following FIFO stack expands to over 100 lines of code when passed through the transformations. The complete transformed design is given in appendix E.

```
TYPE bool = NEW ( t | f | x ),
     int  = NEW i/(0..100).

MAC FIFO {INT size} = (int: data_in,  bool: shift_in shift_out)
                                   -> (int, bool):
(SEQ
      PVAR fifo ::= [size](i/0, f);

      CASE shift_out
         OF  t : fifo := fifo[2..size] CONC (i/0, f)
      ESAC;

      VAR entered := f;
      CASE shift_in
      OF  t : [INT i = 1..size-1]
                  CASE (entered, fifo[i][2])
                    OF  (f,f) : ( fifo[i] := (data_in, t);
                                  entered := t
                                )
                  ESAC
      ESAC;

      OUTPUT (fifo[1][1], entered)
).

FN FIFO_9 = (int: data_in,  bool: shift_in shift_out) -> (int, bool):
FIFO {9} (data_in, shift_in, shift_out).
```

It is possible to synthesis ELLA circuits down to gate level by using one of the currently available synthesis systems, e.g. LOCAM or GATEMAP, appendix F gives an example of a circuit which has passed through the GATEMAP system.

In the next section we examine the different parts of ELLA and define the language features of Core ELLA.

3.3 ELLA Hierarchical Definition

For the purposes of this section the ELLA language will be subdividing into three classes. The first of these classes contains those ELLA constructs for which a formal semantic definition will not be required

3.3.1 Excluded Constructs

There are five language constructs which come into this class and they each have a different reason for being selected

- Attributes (these provide an interface to external software)
- ARITH (being superseded by BIOPs)
- ALIEN (non-ELLA code)
- Machine Dependent BIOPs (Because of machine dependency)
- FNSET (being superseded by extensions to function types)

Only the first four of these apply to the current release of ELLA V4 [Com90a]. The function type extension [HWM91] allows function type signals in the output part of a function specification, this means that function sets can be represented by functions with function type output signals and therefore they need not be considered as a separate case.

It is probable that all the operations which could be performed with ARITH statements will be achievable through the BIOP mechanism and therefore there is no loss of functionality by including ARITH in this class of constructs. The remaining three constructs either describe non-ELLA language code or code which depends on the platform on which ELLA is installed.

3.3.2 Core ELLA Constructs

This second class contains those constructs which will form Core ELLA. It would have been possible to reduce the size of Core ELLA even further if functions had been transformed out and only basic enumerated types allowed. However this would have left only a single large network of nodes for any ELLA context and such a network, whilst suitable for a simulator, would have been very unhelpful for a user of formal methods. In particular if a user experimented with rules built up from the operational semantics there would be a problem in relating information back to the user, such as what part of the original ELLA circuit was being analysed.

All the basic language primitives have been incorporated including the latest additions to the ELLA language i.e. those from the numerics package [HPC$^+$90], retiming [HWM90a] and BIOP's [Tai88b]. Rows have been included since they provide useful functionality and they complement STRING's. Constant, integer and macro declarations are not included since they are evaluated before a circuit is assembled via the transformation system. Only global type and function declarations have been included in order to keep the typing and function environments simple. Single makes, joins and lets are only necessary since the multiple version of these constructs can be transformed into the simple versions.

3.3.2.1 Types

The complete ELLA typing system has been included, i.e.

- Enumerated declaration (e.g. bool, int, char)
- Associated declaration (e.g. tag&bool)
- Unknown value (e.g. ?type)
- Collaterals (e.g. (bool,bool))
- Rows (e.g. [3]bool)
- STRINGS (of ELLA characters e.g. STRING[3]chars)

Thus all forms of enumerated types including ELLA integers and ELLA characters are available in Core ELLA. All possible forms of structured types are also available as well as an ELLA unknown value corresponding to each type.

3.3.2.2 Constants and Integers

Within Core ELLA all constants and integers will be given as explicit basic values. Since constant and integer expressions are always defined statically in ELLA and the **macro** transformation can simplify them to basic values this is not a severe limitation. Hence the type of expressions available are

- Constant enumerated values (e.g. true, i/2, c'z)
- Constant associated type values (e.g. tag&true)
- Integer values (e.g. 5)

3.3.2.3 Primitive Nodes

All the primitive nodes of ELLA are included in Core ELLA. These are

- CASE with no 'ELSEOF' (e.g. CASE bool OF t:f ELSE x ESAC)
- DELAY (e.g. Ambiguity Delay primitive)
- IDELAY (e.g. Inertial Delay primitive)
- RAM (e.g. RAM([256]i/1)).
- SAMPLE (e.g. the SAMPLE-and-hold timing primitive used for retiming)
- BIOP (e.g. machine independent Built-In-Operators)

It is necessary to include all these primitive nodes of ELLA since they are not easily transformed. The CASE statement has been restricted to exclude the ELSEOF alternative since the ELSEOF part can be transformed out. All possible forms of Delays are needed since they are fundamental in providing ELLA's functionality. The new sample-and-hold primitive has been included so that designs with retiming can be transformed into Core ELLA. Only those BIOP's which are machine independent are included here.

3.3.2.4 Functions

Function declarations have been retained within Core ELLA in order to preserve the modularity of a design. It would be possible to transform out all functions within a context but this would leave only a flattened circuit. Most designers use functions in order to partition designs into more manageable parts and this feature is felt to be desirable for Core ELLA. By allowing functions it also means that operations on them can be included e.g. make, join. The complete list of function related constructs is

- Functions Declarations
- MAKE (e.g. MAKE AND: and.)
- Implicit Monadic Function Calls (e.g. AND(in1,in2))
- JOIN (e.g. JOIN ... → name)

- LET (e.g. LET name = ...)
- BEGIN...OUTPUT...END clause

Only monadic function calls have been included since a dyadic call is just another way of representing a monadic function with two inputs. Both the JOIN and LET statements are restricted to the ELLA V4 syntax [Com90a] since the enhancements recently carried out [HWM90c] can be transformed into that form. The BEGIN...END clauses is included in order to allow the LET, JOIN, MAKE statements to be used.

3.3.2.5 Signal Structuring and Extraction

Signals within a Core ELLA program will need to be structured, or extracted, from other signals. For this reason the following have been included in Core ELLA.

- Indexing
- CONC (concatenation)
- REFORM (of signal groups)
- Associated type constructs (i.e. '//' and '&')
- Rows/STRINGS of value delivering clauses
- Collaterals of value delivering clauses

Indexing enables components of structures to be extracted and CONC enables components to be combined. REFORM is a means by which a set of signals can be regrouped to form a different combination e.g. (bool,[3]bool) can be reformed to ([2]bool,[2]bool). The two associated type constructors are need since there is no other way for creating and extracting data from an associated type. The last two items are ways in which structures of value delivering clauses can be constructed.

3.3.3 Transformable Constructs

The last class contains the remainder of the ELLA language. It should be noted that some of the features in this list are not available in the current commercial release of ELLA V4 [Com90a] but they are available in an RSRE version [HWM90c] [HWM91] which will form ELLA V6, see appendix B. These language features are already transformed by software into a series of constructs in Core ELLA (see section 3.2). Thus although at first sight this class of language features might appear to encompass a large amount of ELLA, all language features up to and including those of Release 6 will be transformable into Core ELLA. The one language feature which has not been included in this list is the 'void-type' (or 'null input'), this is because the final range of this type has not been decided.

The complete list of constructs in this transformable class is

- Trimming (e.g. id[1..6])
- IF boolean THEN unit ELSE unit FI
- Named outputs (e.g. FN A = (bool) → (bool:out):...)

- Multiple lets (e.g. LET (a1, a2) = ...)
- Multiple makes (e.g. MAKE [2][3][2]A: a.)
- Complex joins (e.g. JOIN (true,false) → (a1, a2[2]).)
- Replicated joins (e.g. FOR INT i = 1..2 JOIN ...)
- Replicators ([INT i = 1 .. n])
- Dynamic Indexing (indexing by a signal e.g. structure[[signal]])
- ELSEOF (extended CASE statements)
- INT declarations (e.g. INT i = 6)
- CONST declarations (e.g. CONST c = (true, (true | false)).)
- Character ranges (e.g. (‘a .. ‘d) replaced by (‘a | ’b | ’c | ’d))
- Sequences (e.g. VAR, PVAR etc.)
- Macros (with INT, TYPE, CONST, FN, MAC parameters)
- Function types (bidirectional wires e.g. fntype = bool → [2]bool)
- IO (supplying the Input and Output of a function type: used in ‘joining’)
- Locally declared functions, macros, types, ints, consts
- Implicit dyadic function instantiations (e.g. in1 AND in2)
- FASTER/SLOWER (hierarchical retiming constructs e.g. speed up/slow down)
- Print/Fault (macro expansion assertions)
- Imports/exports (control of functions through multiple contexts)
- Renamed (renaming of imported function)
- Static operators (e.g. int/(1+(m*20))

Having divided ELLA into the three classes given above only the second class, Core ELLA, will be considered in the remainder of this chapter. In the next section a set of syntactic classes and definitions are given for Core ELLA.

3.4 Core ELLA Composite Syntax

As defined in [BHM90] a composite syntax combines both concrete and abstract syntax of a language. The composite syntax has been defined to retain the binding of the concrete syntax, thus it accurately represents the binding of the ELLA parser. The composite syntax defined in this section is summarised in appendix C .

In order to define the composite syntax a set of conventions and syntactic categories are need. Their definition has been chosen in order to maximise the readability of the syntax. Before proceeding to define the conventions some basic notation will be given

3.4.1 Basic Notation

Throughout this section the following notation will be used

abc $\in$ Abc	$\equiv$	'abc' is an element of the set 'Abc'
b ::= c	$\equiv$	the syntax definition of 'b' is 'c'
$\vert$	$\equiv$	the separator of alternatives in a syntax definition
$\vert$	$\equiv$	ELLA separator of alternatives
$d_1 \cdots d_k$	$\equiv$	one or more occurrences of 'd'
$d_1, \cdots, d_k$	$\equiv$	one or more occurrences of 'd' separated by ','. Note if k=1 then no ',' is present.
$d_1, \cdots, d_{k\text{-}1}$	$\equiv$	zero or more occurrences of 'd' separated by ','. Note if k=0 then no ',' is present.

3.4.2 Conventions

The conventions used throughout this section are described below. In these conventions words which begin with an upper case letter represent Sets, where 'Identifier' is the set of all lower case names, 'Fnname' the set of all upper case names and symbols. Note that 'Identifier' and 'Fnname' are disjoint Sets. 'Character' is the Set of all printable characters, whilst 'String' is the set of all character strings composed from elements of 'Character'. The Sets '$\mathbb{Z}$' and '$\mathbb{N}$' are the Sets of integers and natural numbers respectively.

typename	$\in$	Identifier	(ELLA type name e.g. lower case)
signalname	$\in$	Identifier	(ELLA signal name)
tagname	$\in$	Identifier	(ELLA tagged type name)
altname	$\in$	Identifier	(ELLA enumerated type alternative)
fnname	$\in$	Fnname	(ELLA function name e.g. upper case or symbol)
biopname	$\in$	Fnname	(ELLA BIOP name e.g. upper case)
z	$\in$	$\mathbb{Z}$	(An integer)
lwb, upb	$\in$	$\mathbb{Z}$	(An integer)
j,k	$\in$	$\mathbb{N}_1$	(A non-zero positive integer)
index	$\in$	$\mathbb{N}_1$	(A non-zero positive integer)
size	$\in$	$\mathbb{N}_1$	(A non-zero positive integer)
interval	$\in$	$\mathbb{N}_1$	(ELLA timing interval)
ambig_time	$\in$	$\mathbb{N}_1$	(Ambiguity delay time)
delay_time	$\in$	$\mathbb{N}_1$	(delay time)
skew_time	$\in$	$\mathbb{N}_1$	(skew delay)
con	$\in$	Constant	('con' is a value of the ELLA Constant type 'Constant')
initial_value	$\in$	Constant	(Delay, Retiming or Ram initialisation value)
ambig_value	$\in$	Constant	(Delay ambiguity value)
char	$\in$	Character	(A printable character e.g. 'a')
string	$\in$	String	(A string of printable characters e.g. 'abc')

Some of these names have been taken from the abstract syntax (i.e. delay parameter names) in order to aid readability.

3.4.3 Basic Enumerated Values

The following four basic enumerated values will be grouped together

enumerated	::=	altname
		| tagname / z
		| tagname 'char
		| tagname "string"

The symbols /, ', " are ELLA defined symbol tags which enable the 'tagname' to be correctly associated with the appropriate ELLA type. It can be noted that the basic associated type value is missing from the list since the associated part is either a constant, unit or type.

3.4.4 Types

A type in Core ELLA is defined to be

type	::=	typename
		| STRING [size] type
		| [size] type
		| ($\text{type}_1, \cdots, \text{type}_k$)

where 'typename' is the name of a type declaration (see below). Note that the collateral of types can have just one element which therefore allows redundant brackets in types. The constant 'size' is taken to have a known positive integer value.

3.4.5 Constants

Core ELLA constants have been split into two syntactic classes in order to preserve the binding of the concrete syntax.

const	::=	STRING [size] const1
		| [size] const
		| const1

const1	::=	enumerated
		| tagname & const1
		| ($\text{const}_1, \cdots, \text{const}_k$)
		| ? const1

Constants defined by this syntax will be used as initialising values in delays and rams.

3.4.6 Constant Sets

Constant Sets are collections of constants which are used as choosers in CASE statements. The definition of Constant Sets differs from that of Constants in two ways. First the inclusion of the top level definition which allows for constant alternatives. Second the last option of 'constset2' is 'type' and not '?const1'. The last change is necessary since it is not possible to test for '?const1' in a CASE statement. The full definition of Constant Sets is

constset	::=	$\text{constset}_1 \mid \cdots \mid \text{constset}_k$

constset1	::=	STRING [size] constset2
		| [size] constset1
		| constset2

constset2	::=	enumerated
		| tagname & constset2
		| ($\text{constset}_1, \cdots, \text{constset}_k$)
		| type

3.4.7 Units

The syntax element 'unit' is the basic building block for all Core ELLA descriptions. Units supply single or multiple values and are composed of either basic values or clauses which deliver basic values. The definition given here has three levels which represent the levels of binding given in the concrete syntax.

unit	::=	unit CONC unit1
		| unit1
unit1	::=	STRING [size] unit1
		| [size] unit1
		| fnname unit1
		| tagname & unit1
		| unit2 // tagname
		| unit2
unit2	::=	signalname
		| enumerated
		| unit2 [index]
		| ? const1
		| closedclause

The last alternative of 'unit2' is a closedclause which is defined in the next section. The 'signalname' of 'unit2' is either a MAKE, LET or parameter name. The constant 'index' is a known integer value.

3.4.8 Closedclause

Closedclauses provide the mechanism for combining units. The CASE statement provides a multiplexer-type construct, and the BEGIN...END clause gives a method for signal naming and introduces LET, MAKE or JOIN.

closedclause	::=	CASE unit OF cases ELSE unit ESAC
		| ($\text{unit}_1, \cdots, \text{unit}_k$)
		| BEGIN $\text{step}_1 \cdots \text{step}_{k\text{-}1}$ OUTPUT unit END
cases	::=	$\text{constset}_1 : \text{unit}_1, \cdots, \text{constset}_k : \text{unit}_k$
step	::=	LET $\text{letdecs}_1, \cdots, \text{letdecs}_k$.
		| MAKE $\text{makedecs}_1, \cdots, \text{makedecs}_k$.
		| JOIN $\text{joindecs}_1, \cdots, \text{joindecs}_k$.
letdecs	::=	signalname = unit
makedecs	::=	fnname : signalname
joindecs	::=	unit $\rightarrow$ signalname

Note that within a BEGIN $\cdots$ END clause there need not be any 'step's. Hence step$_{k\text{-}1}$ implies that if k = 1 then no 'step's are present. It should be noted that although each of the 'step' declarations have a 'signalname' in them there is a difference in the way they are handled. For example the 'signalname' in a LET names the output of the 'unit' and hence it can only be used to supply a signal to some other part of a circuit. This means that such a signalname could not occur in the right hand side of a 'joindecs'. Whereas the signalname in the makedecs is naming a function instantiation and hence has both a value-requiring part and a value-delivering part. This means such a signalname could be used on either side of the '$\rightarrow$' in a joindecs.

3.4.9 Function Body

A function body can either be described by a 'unit' or by a function body primitive.

functionbody	::=	unit
		| REFORM
		| BIOP biopname
		| DELAY (initial_value, ambig_time, ambig_value, delay_time)
		| IDELAY (initial_value, delay_time)
		| SAMPLE (interval, initial_value, skew_time)
		| RAM (initial_value)

All the basic syntactic definitions of Core ELLA have now been completed. The three remaining classes define how the above information is combined to form a Core ELLA closure.

3.4.10 Type Declaration

A Type declaration is of the following form

typedec	::=	TYPE typename = typeornew.
typeornew	::=	type
		| NEW tagname / (lwb .. upb)
		| NEW (typealt$_1$ $\vert\cdots\vert$ typealt$_k$)
		| NEW tagname ('char$_1$ $\vert\cdots\vert$ 'char$_k$)
typealt	::=	tagname & type
		| altname

where the alternatives of 'typeornew' represent: the name of a previous type declaration, an ELLA integer, an enumerated type, and an ELLA character. The enumerated type allows for some of the alternatives to have associated values.

3.4.11 Function Declaration

A function declaration is given by

fndec	::=	FN fnname = (input) $\rightarrow$ type : functionbody.
input	::=	type$_1$: signalname$_1$, $\cdots$, type$_k$: signalname$_k$

3.4.12 Closure

A Core ELLA Closure is the entry point into a Core ELLA context and consists of a number of type and function declarations.

closure ::= $\text{typedec}_1 \cdots \text{typedec}_j\ \text{fndec}_1 \cdots \text{fndec}_k$

In any closure function declarations are built upon previously defined function declarations. Thus fndec_i can only use function declarations fndec_l where $1 \leq l < i$. Hence fndec_k is considered to be the top level function declaration.

Chapter 4

Kernel ELLA

This chapter contains the definition of **Kernel** ELLA. The **Kernel** is a set of data structures which can, by means of a set of transformation rules, express circuits defined in the Core. The transformation rules from the Core to the **Kernel** remove scopes and type checking, thus making the **Kernel** a more mathematically convenient system for the purposes of semantic definition and the provision of verification. Accordingly, this chapter concentrates on specifying the transformation of Core ELLA into Kernel ELLA and as such provides a static semantics for the Core. An example of the use of the transformation rules is presented.

4.1 Introduction

In this chapter we present the **Kernel** of ELLA. The **Kernel** is a set of data structures which can describe the complete ELLA language. The **Kernel** is not itself a language since it does not have the problems of scopes or type checking; these are all taken care of by the transformations from Core ELLA to the **Kernel**. Certain subsets of the **Kernel** can be defined to have a concrete syntax and in the next chapter an example of such a subset is given.

The aims of the **Kernel** are that it should provide a simple system for mathematical manipulation and that it should interface simply with the verification work. Although all the scopes have been removed, function hierarchy has not. This means that a design can retain its structure. In the next section we describe the **Kernel** data structures, with the transformations from Core ELLA to the **Kernel** being dealt with in subsequent sections. The complete **Kernel** data structures are summarised in appendix D

The work in this chapter is still under development and some aspects, in particular the checking mechanism, still need completing. In addition minor alterations to the data structures may still be necessary, in particular, one due to a new language feature under development. At present no mechanical checks on the transformations have been done and therefore their accuracy still remains to be proved. This work will form the next stage of the project.

4.2 Conventions

Throughout this chapter the following conventions will be observed:-

abc	$\in$	Abc (ie. it is an element of the set Abc)
Indexer, Size, Fnno	$\subseteq$	$\mathbb{N}_1$
Typeno, Tagno, Inputno	$\subseteq$	$\mathbb{N}_1$

Signalno, Ambigtime, Delaytime	$\subseteq$	$\mathbb{N}_1$
Interval, Skew	$\subseteq$	$\mathbb{N}_1$
Inputtype, Outputtype	$\subseteq$	Type
Initialconst, Ambigconst	$\subseteq$	Const
Fnname, Biopname	$\subseteq$	"Upper case identifier or operator"
Name, Signalname	$\subseteq$	"Lower case identifier"
Typename, Tagname	$\subseteq$	"Lower case identifier"
Lowerbound, Upperbound	$\subseteq$	"positive or negative integer"
Character	$\subseteq$	"printable character"
kId	$\triangleq$	"object "Id" belongs in the **Kernel**"
a.b	$\triangleq$	"Field selector, (i.e. field "b" of "a")"
a[int]	$\triangleq$	"Indexing, (i.e. element "int" of "a")"

In the following definition of the **Kernel** data structure, the following naming conventions are used:

abc	:	Abc	the value 'abc' is an element of the set Abc
$Abc_1, Abc_2, Abc_3, \cdots$	=	Abc	separate instances of tuple Abc
$AbcSeq$	=	$Abc \times \cdots \times Abc$	non-empty sequence of elements of type Abc
$AbcOpt$	=	$Abc \mid \{$**nil**$\}$	type Abc with optional element **nil**

One of the reasons for adopting these conventions is that names of component types are used as field selectors for structured types. This means that every field needs to be indicated by a unique name.

It can be noted that the data structures definitions below naturally correspond to type definitions in the style of VDM. This correspondence is illustrated by means of the following extract of the **Kernel**

Enumerated	::=	Enum
		| **string**(Typeno $\times$ TagnoSeq)
Enum	::=	**enum**(Typeno $\times$ Tagno)
Closure	::=	TypedecSeq $\times$ FndecSeq

which corresponds to the following in VDM:

$$Enumerated = Enum \mid String$$

$$\begin{aligned} String :: \quad typeno &: Typeno \\ tagnoseq &: Tagno^+ \end{aligned}$$

$$\begin{aligned} Enum :: \ typeno &: Typeno \\ tagno &: Tagno \end{aligned}$$

$$Closure = Typedec^+ \times Fndec^+$$

4.3 Kernel of ELLA Data Structure

In this section we describe the data structure model of the **Kernel**.

Note that all names in the **Kernel** are referenced by integers. These are unique even though the scopes have been removed and their identifiers are no longer unique. These integers index the first three fields of the environment and the names are merely attributes of the indexed objects. For example ‘Signalno’ in **signal** indexes the sigdec field of the environment and the original name is obtained from the Signalname field of this object.

4.3.1 Enumerated

Enumerated values in the **Kernel** are defined as

Enumerated	::=	Enum
		$\mid$ **string**(Typeno $\times$ TagnoSeq)
Enum	::=	**enum**(Typeno $\times$ Tagno)

where Enum represents all the different enumerated types except for STRING types.

4.3.2 Types

Types in the **Kernel** are defined by the following structures

Type	::=	**typeno**(Typeno)
		$\mid$ **typename**(Typename $\times$ Type)
		$\mid$ **stringtype**(Size $\times$ Typeno)
		$\mid$ **types**(TypeSeq)
Typety	::=	TypeOpt

4.3.3 Constants

Constants are the representation of the constant class of Core ELLA and are defined by

Const	::=	Enumerated
		$\mid$ **conststring**(Size $\times$ Const)
		$\mid$ **consts**(ConstSeq)
		$\mid$ **constassoc**(Enum $\times$ Const)
		$\mid$ **constquery**(Type)

4.3.4 Constant Sets

Constant sets are similar to constants except that they allow sets of alternatives (i.e. **true**| **false**)

Constset	::=	Enumerated
		$\mid$ **constsetalts**(ConstsetSeq)
		$\mid$ **constsetstring**(Size $\times$ Constset)
		$\mid$ **constsets**(ConstsetSeq)
		$\mid$ **constsetassoc**(Enum $\times$ Constset)
		$\mid$ **constsetany**(Type)

4.3.5 Units

The Unit expressions are defined by

Unit ::= Enumerated
| **conc**(Unit $\times$ Unit $\times$ Outputtype)
| **unitstring**(Size $\times$ Unit)
| **units**(UnitSeq)
| **call**(Fnno $\times$ Unit)
| **unitassoc**(Enum $\times$ Unit)
| **restrict**(Unit $\times$ Enum $\times$ Outputtype)
| **signal**(Signalno)
| **index**(Unit $\times$ Indexer $\times$ Outputtype)
| **unitquery**(Type)
| **caseclause**(Unit $\times$ CaseSeq $\times$ UnitOpt $\times$ Outputtype)

where the different alternatives in Case clauses are given by

Case ::= **case**(Constset $\times$ Unit)

4.3.6 Function Declarations

A function declaration has the following data structure

Fndec ::= **fndec**(Fnname $\times$ Inputtype $\times$ SignaldecSeq $\times$ Outputtype $\times$ Fnbody)

where

Signaldec ::= **signaldec**(Signalname $\times$ Type $\times$ Unit | **input**)

and the Function Body is given by

Fnbody ::= Unit
| **reform**
| **biop**(Biopname)
| **delay**(Initialconst $\times$ Ambigtime $\times$ Ambigconst $\times$ Delaytime)
| **idelay**(Initialconst $\times$ Delaytime)
| **sample**(Interval $\times$ Initialconst $\times$ Skew)
| **ram**(Initialconst)

4.3.7 Type Declarations

A type declarations are of the form

Typedec ::= **typedec**(Typename $\times$ New)

where

New ::= **tags**(TagSeq)
| **ellaint**(Tagname $\times$ Lowerbound $\times$ Upperbound)
| **chars**(Tagname $\times$ CharacterSeq)

Tag ::= **tag**(Tagname $\times$ TypeOpt)

4.3.8 Closures

Finally a closure is represented by

Closure ::= TypedecSeq $\times$ FndecSeq

4.4 Environments and Signatures

This section describes the environment of the **Kernel** and the signatures on Core ELLA, these being needed for the transformations from Core ELLA to the **Kernel**.

4.4.1 Environment

The environment (Env) is defined to be a record object with 10 fields which will accumulate type, function and signal declarations and maintain information about the scopes of identifiers.

$$
\begin{array}{lrcl}
\mathit{Env} :: & \mathit{typedec} & : & \mathit{kTypedec}^* \\
 & \mathit{fndec} & : & \mathit{kFndec}^* \\
 & \mathit{sigdec} & : & \mathit{kSignaldec}^* \\
 & \mathit{fnmap} & : & \mathit{Fnname} \longrightarrow \mathit{Fnno} \\
 & \mathit{lclfnmap} & : & \mathit{Fnname} \longrightarrow \mathit{Fnno} \\
 & \mathit{namemap} & : & \mathit{Name} \longrightarrow \mathit{Typetag} \\
 & \mathit{lclnamemap} & : & \mathit{Name} \longrightarrow \mathit{Typetag} \\
 & \mathit{signamemap} & : & \mathit{Signalname} \longrightarrow \mathit{Sig} \\
 & \mathit{lclsignamemap} & : & \mathit{Signalname} \longrightarrow \mathit{Sig} \\
 & \mathit{usedsigname} & : & \mathit{Signalname\text{-}set}
\end{array}
$$

$$\mathit{inv}(E) \quad \triangleq \quad \mathbf{dom}\,(E.\mathit{lclnamemap}) \cap \mathbf{dom}\,(E.\mathit{lclsignamemap}) \cap \mathbf{dom}\,(E.\mathit{signamemap}) = \{\,\}$$

with the following being local to the translation process

$$
\begin{array}{lcl}
\mathit{Typetag} & = & \mathbf{typeno}(\mathit{Typeno}) \cup \ \mathbf{typename}(\mathit{Typename} \times \mathit{kType}) \cup \ \mathbf{consttag}(\mathit{Typeno}) \\
\mathit{Sig} & = & \mathbf{sig}(\mathit{Signalno} \times \mathit{kType} \times \mathit{Sort}) \\
\mathit{Sort} & = & \mathbf{joined} \mid \ \mathbf{unjoined}
\end{array}
$$

Note that the first three fields of E are sequences. The use of each field can be summarised as follows

E.typedec	Accumulates typedecs for the final closure
E.fndec	Accumulates fndecs for the final closure
E.sigdec	Accumulates signaldecs for each fndec
E.lclfnmap, *E.lclnamemap*, *E.lclsignamemap*	Names in the most local scopes,
E.fnmap, *E.namemap*, *E.signamemap*	Names visible outside the most local scope,
E.usedsigname	Names out of scope but unavailable for reuse

Note Fnname's are generated by FN declarations, Name's are generated by TYPE declarations (both the TYPE name and the tags), and Signalname's are generated by MAKE, LET and input parameter declarations. The field 'usedsigname' is the set of all signal names which have been used in the current FN but are now out of scope. This field is used by Check-Signal to disallow their redefinition. This is a deliberate language restriction to ensure unique signal names in function bodies.

The initial environment contains only empty declarations and scopes i.e.

$$\mathit{InitialEnv} = \ \mathbf{env}([\,],[\,],[\,],\{\,\},\{\,\},\{\,\},\{\,\},\{\,\},\{\,\},\{\,\})$$

4.4.2 Signatures

Throughout this chapter a number of signatures will be needed. The complete list is given by

Scope-Fn-Begin	:	Env $\longrightarrow$ Env
Scope-Fn-End	:	Env $\times$ Env $\longrightarrow$ Env
Scope-Begin	:	Env $\longrightarrow$ Env
Scope-End	:	Env $\times$ Env $\longrightarrow$ Env
Check-Joins	:	Env $\longrightarrow \mathbb{B}$
Check-Two-Val	:	kType $\longrightarrow \mathbb{B}$
Check-Fn	:	Fnname $\times$ Env $\longrightarrow \mathbb{B}$
Check-Name	:	Name $\times$ Env $\longrightarrow \mathbb{B}$
Check-Signal	:	Signalname $\times$ Env $\longrightarrow \mathbb{B}$
Add-Fn	:	Fndec $\times$ Env $\longrightarrow$ Env
Add-Type	:	Typedec $\times$ Env $\longrightarrow$ Env
Add-Signal	:	Signaldec $\times$ Sort $\times$ Env $\longrightarrow$ Env
Add-Join	:	Signaldec $\times$ Signalno $\times$ Env $\longrightarrow$ Env
Add-Tag	:	Tagname $\times$ Env $\longrightarrow$ Env
Add-Type-Name	:	Typename $\times$ kType $\times$ Env $\longrightarrow$ Env
Find-Nm	:	Name $\times$ Env $\longrightarrow$ Typetag $\cup$ Sig
Find-Fn	:	Fnname $\times$ Env $\longrightarrow$ Fnno
Find-Unjoined	:	Signalname $\times$ Env $\longrightarrow$ Fnno
Find-Type	:	Typename $\times$ Env $\longrightarrow$ kType
Find-Alt	:	Altname $\times$ Env $\longrightarrow$ kEnum
Find-ELLAint	:	Tagname $\times$ Env $\longrightarrow$ Typeno $\times$ Lowerbound $\times$ Upperbound
Find-Integer-Type	:	kType $\times$ Env $\longrightarrow$ Typeno $\times$ Lowerbound $\times$ Upperbound
Find-Char	:	Tagname $\times$ Char $\times$ Env $\longrightarrow$ kEnum
Find-Signal	:	Signal $\times$ Env $\longrightarrow$ kUnit $\times$ kType
Find-Assoc	:	Altname $\times$ Env $\longrightarrow$ kTypeOpt
Is-Char	:	$\mathbb{N}_1 \times$ Env $\longrightarrow \mathbb{B}$
Index	:	kType $\times \mathbb{N}_1 \longrightarrow$ kType
Flatten	:	kType $\longrightarrow$ kTypeSeq
TypeTuple	:	kTypeSeq $\longrightarrow$ kType
ConstTuple	:	kConstSeq $\longrightarrow$ kConst
ConstsetTuple	:	kConstsetSeq $\longrightarrow$ kConstset
UnitTuple	:	kUnitSeq $\longrightarrow$ kUnit
Disjoint	:	kConstset $\times$ kConstset $\longrightarrow \mathbb{B}$
Conc	:	kType $\times$ kType $\longrightarrow \mathbb{B}$
KERNEL	:	Closure $\longrightarrow$ kClosure

$$
\begin{array}{lcl}
\_ \vdash \_ \dashv\boxed{\mathrm{EM}}\Rightarrow \_ & \subseteq & \mathit{Env} \times \mathit{Enum} \times \mathit{kEnum} \\
\_ \vdash \_ \dashv\boxed{\mathrm{T}}\Rightarrow \_ & \subseteq & \mathit{Env} \times \mathit{Type} \times \mathit{kType} \\
\_ \vdash \_ \dashv\boxed{\mathrm{C}}\Rightarrow \_ : \_ & \subseteq & \mathit{Env} \times \mathit{Const} \times \mathit{kConst} \times \mathit{kType} \\
\_ \vdash \_ \dashv\boxed{\mathrm{CS}}\Rightarrow \_ : \_ & \subseteq & \mathit{Env} \times \mathit{Constset} \times \mathit{kConstset} \times \mathit{kType} \\
\_ \vdash \_ \dashv\boxed{\mathrm{U}}\Rightarrow \_ : \_ , \_ & \subseteq & \mathit{Env} \times \mathit{Unit} \times \mathit{kUnit} \times \mathit{kType} \times \mathit{Env} \\
\_ \vdash \_ \dashv\boxed{\mathrm{CC}}\Rightarrow \_ : \_ , \_ & \subseteq & \mathit{Env} \times \mathit{Closedclause} \times \mathit{kUnit} \times \mathit{kType} \times \mathit{Env} \\
\_ \vdash \_ \dashv\boxed{\mathrm{CA}}\Rightarrow \_ , \_ : \_ , \_ , \_ & \subseteq & \mathit{Env} \times \mathit{Constset} \times \mathit{kConstset} \times \mathit{kUnit} \times \mathit{kType} \times \mathit{kType} \times \mathit{Env} \\
\_ \vdash \_ \dashv\boxed{\mathrm{SP}}\Rightarrow \_ & \subseteq & \mathit{Env} \times \mathit{Step} \times \mathit{Env} \\
\_ \vdash \_ \dashv\boxed{\mathrm{NW}}\Rightarrow \_ , \_ & \subseteq & \mathit{Env} \times \mathit{New} \times \mathit{kNew} \times \mathit{Env} \\
\_ \vdash \_ \dashv\boxed{\mathrm{TA}}\Rightarrow \_ , \_ & \subseteq & \mathit{Env} \times \mathit{Typealt} \times \mathit{Tag} \times \mathit{Env} \\
\_ \vdash \_ \dashv\boxed{\mathrm{TD}}\Rightarrow \_ & \subseteq & \mathit{Env} \times \mathit{Typedec} \times \mathit{Env} \\
\_ \vdash \_ \dashv\boxed{\mathrm{FD}}\Rightarrow \_ & \subseteq & \mathit{Env} \times \mathit{Fndec} \times \mathit{Env} \\
\_ \vdash \_ \{\_ \_\} \dashv\boxed{\mathrm{BI}}\Rightarrow \_ & \subseteq & \mathit{Env} \times \mathit{Builtin} \times \mathit{kType} \times \mathit{kType} \times \mathit{kBuiltin} \\
\_ \vdash \_ \dashv\boxed{\mathrm{IN}}\Rightarrow \_ , \_ & \subseteq & \mathit{Env} \times \mathit{Inputtype} \times \mathit{kType} \times \mathit{Env} \\
\\
\_ \vdash \_ \dashv\boxed{\mathrm{CL}}\Rightarrow \_ & \subseteq & \mathit{Env} \times \mathit{Closure} \times \mathit{Env} \\
\\
\_ \dashv\boxed{\mathrm{T}}\vDash \_ & \subseteq & \mathit{kType} \times \mathit{kType}
\end{array}
$$

4.4.3 Scopes

The scopes of Core ELLA are removed by the transformation to the **Kernel**. In order to achieve this the following are needed

$\mathit{Scope\text{-}Fn\text{-}Begin}: \mathit{Env} \to \mathit{Env}$

$$
\begin{array}{l}
\mathit{Scope\text{-}Fn\text{-}Begin}(E) \mathrel{\underline{\triangle}} \\
\quad \mathbf{env}(\; \begin{array}[t]{lll}
E.\mathit{typedec}, & E.\mathit{fndec}, [\,], & \\
E.\mathit{fnmap} \dagger E.\mathit{lclfnmap}, & \{\,\}, & E.\mathit{namemap} \dagger E.\mathit{lclnamemap}, \\
\{\,\}, & \{\,\}, & \{\,\} \\
\{\,\}) & &
\end{array}
\end{array}
$$

$\mathit{Scope\text{-}Fn\text{-}End}: \mathit{Env} \times \mathit{Env} \to \mathit{Env}$

$$
\begin{array}{l}
\mathit{Scope\text{-}Fn\text{-}End}(E, L) \mathrel{\underline{\triangle}} \\
\quad \mu(E,\; \{\mathit{typedec} \mapsto L.\mathit{typedec},\; \mathit{fndec} \mapsto L.\mathit{fndec}\})
\end{array}
$$

$\mathit{Scope\text{-}Begin}: \mathit{Env} \to \mathit{Env}$

$$
\begin{array}{l}
\mathit{Scope\text{-}Begin}(E) \mathrel{\underline{\triangle}} \\
\quad \mathbf{env}(\; \begin{array}[t]{lll}
E.\mathit{typedec}, & E.\mathit{fndec}, & E.\mathit{sigdec}, \\
E.\mathit{fnmap} \dagger E.\mathit{lclfnmap}, & \{\,\}, & E.\mathit{namemap} \dagger E.\mathit{lclnamemap}, \\
\{\,\}, & E.\mathit{signamemap} \dagger E.\mathit{lclsignamemap}, \{\,\} & \\
E.\mathit{usedsigname}) & &
\end{array}
\end{array}
$$

$Scope\text{-}End : Env \times Env \rightarrow Env$

$Scope\text{-}End(E, L) \triangleq$
$\quad$ **let** $E_1 = Scope\text{-}Fn\text{-}End(E, L)$ **in**
$\quad\quad \mu(E_1,\ usedsigname \mapsto (E.usedsigname \cup \mathbf{dom}\,(L.lclsignamemap)))$

Although *Scope-Fn-Begin* and *Scope-Fn-End* are not actually needed for the **Kernel**, since Core ELLA does not presently have any nested functions, they have been included here to allow for expansion of Core ELLA. The stacking and unstacking of the scopes for BEGIN..END clauses is carried out through the transformation rule [CC3].

The scopes work in the following way when transformation rule [CC3] is applied: When a BEGIN is encountered, each local map in the new environment is set to null, and each non-local map is set to the map constructed by combining the local and non-local maps of the previous environment. In the case of common arguments, the local entry overrides the non-local one. Then throughout the ensuing BEGIN..END clause the local map gets made up again with items that are local to the BEGIN..END clause. At the END the local function names and type names will go out of scope and the declarations that were local prior to entry to the BEGIN..END come back into local scope. Thus on leaving the BEGIN..END the new environment takes the first three fields from the local environment and all the other fields from the environment prior to the BEGIN..END, except for those local signal names which are now out of scope but which need to be added to the usedsigname field to ensure that they are not redefined.

4.4.4 Join Checks

The *Check-Joins* predicate is used to ensure that all local signals in an Environment have been joined.

$Check\text{-}Joins : Env \rightarrow \mathbb{B}$

$Check\text{-}Joins(E) \quad \triangleq \quad \forall s \in \mathbf{rng}\, E.lclsignamemap \cdot s.sort = \ \mathbf{joined}$

4.4.5 Check names

These predicates ensure that a particular name is not already in scope. They will be used by the functions that add names to an Environment.

$Check\text{-}Fn : Fnname \times Env \rightarrow \mathbb{B}$

$Check\text{-}Fn(fnname, E) \triangleq$
$\quad fnname \notin \mathbf{dom}\,(E.lclfnmap)$

$Check\text{-}Name : Name \times Env \rightarrow \mathbb{B}$

$Check\text{-}Name(name, E) \triangleq$
$\quad name \notin (\mathbf{dom}\,(E.lclnamemap) \cup \mathbf{dom}\,(E.lclsignamemap))$

$Check\text{-}Signal : Signalname \times Env \rightarrow \mathbb{B}$

$Check\text{-}Signal(signalname, E) \triangleq$

$\quad signalname \notin (\mathbf{dom}\,(E.lclnamemap) \cup \mathbf{dom}\,(E.signamemap)$
$\qquad \cup\, \mathbf{dom}\,(E.lclsignamemap) \cup \mathbf{dom}\,(E.usedsigname))$

4.4.6 Adding Names to an Environment

These functions define the addition of names to an environment

$Add\text{-}Fn : Fndec \times Env \rightarrow Env$

$Add\text{-}Fn(fd, E) \triangleq$

$\quad \mathbf{let}\ Len = \mathbf{len}\, E.fndec\ \ \mathbf{in}$
$\quad \mathbf{let}\ FnName = fd.fnname\ \ \mathbf{in}$
$\quad\ \mu(E, \{\, fndec \mapsto E.fndec \frown [fd],$
$\qquad\quad lclfnmap \mapsto (E.lclfnmap \dagger \{FnName \mapsto Len + 1\})$
$\qquad \}$
$\quad)$

pre

$\quad Check\text{-}Fn(fd.fnname, E)$

$Add\text{-}Type : Typedec \times Env \rightarrow Env$

$Add\text{-}Type(td, E) \triangleq$

$\quad \mathbf{let}\ Len = \mathbf{len}\, E.typedec\ \ \mathbf{in}$
$\quad \mathbf{let}\ TyName = td.typename\ \ \mathbf{in}$
$\quad\ \mu(E, \{\, typedec \mapsto E.typedec \frown [td],$
$\qquad\quad lclnamemap \mapsto (E.lclnamemap \dagger \{TyName \mapsto\ \mathbf{typeno}(Len + 1)\})$
$\qquad \}$
$\quad)$

pre

$\quad Check\text{-}Name(td.typename, E)$

$Add\text{-}Signal : Signaldec \times Sort \times Env \rightarrow Env$

$Add\text{-}Signal(sd, sort, E) \triangleq$

$\quad \mathbf{let}\ Len = \mathbf{len}\, E.sigdec\ \ \mathbf{in}$
$\quad \mathbf{let}\ SigName = sd.signalname\ \ \mathbf{in}$
$\quad\ \mu(E, \{\, sigdec \mapsto E.sigdec \frown [sd],$
$\qquad\quad lclsignamemap \mapsto (E.lclsignamemap \dagger \{SigName \mapsto$
$\qquad\quad\ \mathbf{sig}(Len + 1, sd.type, sort)\})$
$\qquad \}$
$\quad)$

pre

$\quad Check\text{-}Signal(sd.signalname, E)$

$Add\text{-}Join: Signaldec \times Signalno \times Env \to Env$

$Add\text{-}Join(sd, signalno, E) \underline{\triangle}$
let $SigName = sd.signalname$ **in**
let $SigType = sd.type$ **in**
$\mu(E, \{\ sigdec \mapsto (E.sigdec \dagger \{signalno \mapsto sd\}),$
$lclsignamemap \mapsto (E.lclsignamemap \dagger \{SigName \mapsto$
$\mathbf{sig}(signalno,\ SigType,\ \mathbf{joined})\})$
$\}$
$)$

$Add\text{-}Tag: Tagname \times Env \to Env$

$Add\text{-}Tag(tagname, E) \underline{\triangle}$
let $Len =$ **len** $E.typedec$ **in**
$\mu(E, lclnamemap \mapsto (E.lclnamemap \dagger \{tagname \mapsto\ \mathbf{consttag}(Len + 1)\}))$
pre
$Check\text{-}Name(tagname, E)$

$Add\text{-}Type\text{-}Name: Typename \times kType \times Env \to Env$

$Add\text{-}Type\text{-}Name(typename, ktype, E) \underline{\triangle}$
$\mu(E, lclnamemap \mapsto (E.lclnamemap \dagger \{typename \mapsto\ \mathbf{typename}(typename, ktype)\}))$
pre
$Check\text{-}Name(typename, E)$

4.4.7 Finding Names in an Environment

These functions describe how any name can be located within an Environment

$Find\text{-}Nm: Name \times Env \to Typetag \cup Sig$

$Find\text{-}Nm(name, E) \underline{\triangle}$
$(E.namemap \dagger E.lclnamemap\ \dagger E.lclsignamemap)(name)$

$Find\text{-}Fn: Fnname \times Env \to Fnno$

$Find\text{-}Fn(name, E) \underline{\triangle}$
$(E.fnmap \dagger E.lclfnmap)(name)$

$Find\text{-}Unjoined: Signalname \times Env \rightarrow Fnno$

$Find\text{-}Unjoined(signalname, E) \underline{\triangle}$
  let $Signo = (E.lclsignamemap)(signalname).signalno$ **in**
  let signaldec$(signalname, type,$ **call**$(fnno, \_)) = (E.sigdec)[Signo]$ **in**
    $fnno$
pre
  $(E.lclsignamemap)(signalname).sort =$ **unjoined**

$Find\text{-}Type: Typename \times Env \rightarrow kType$

$Find\text{-}Type(typename, E) \underline{\triangle}$
  $Find\text{-}Nm(typename, E)$
pre
  $Find\text{-}Nm(typename, E) \in ($ **typename** $\cup$ **typeno**$)$

$Find\text{-}Alt: Altname \times Env \rightarrow kEnum$

$Find\text{-}Alt(altname, E) \underline{\triangle}$
  let consttag$(ktypeno) = Find\text{-}Nm(altname, E)$ **in**
  let typedec$(\_,$ **tags**$(tags)) = (E.typedec)[ktypeno]$ **in**
  let $index = \iota(i \in$ **inds** $tags) \cdot tags[i] =$ **tag**$(altname, \_)$ **in**
    enum$(ktypeno,\ index)$
pre
  $Find\text{-}Nm(altname, E) \in$ **consttag**

$Find\text{-}ELLAint: Tagname \times Env \rightarrow Typeno \times Lowerbound \times Upperbound$

$Find\text{-}ELLAint(tagname, E) \underline{\triangle}$
  let consttag$(ktypeno) = Find\text{-}Nm(tagname, E)$ **in**
  let typedec$(\_,$ **ellaint**$(\_, lb, ub)) = (E.typedec)[ktypeno]$ **in**
    $ktypeno,\ lb,\ ub$
pre
  $Find\text{-}Nm(tagname, E) \in$ **consttag**

$Find\text{-}Integer\text{-}Type: kType \times Env \rightarrow Typeno \times Lowerbound \times Upperbound$

$Find\text{-}Integer\text{-}Type(ktype, E) \underline{\triangle}$
  cases $ktype$ **of**
   typeno$(typeno) \rightarrow$ **cases** $(E.typedec)[typeno]$ **of**
      typedec$(\_,$ **ellaint**$(t, l, u)) \rightarrow t,\ l,\ u$
     end
   typename$(\_, type) \rightarrow Find\text{-}Integer\text{-}Type(type)$
  end

$Find\text{-}Char: Tagname \times Char \times Env \rightarrow kEnum$

$Find\text{-}Char(tagname, char, E) \triangleq$
  let **consttag**$(ktypeno) = Find\text{-}Nm(tagname, E)$ **in**
  let **typedec**$(\_,$ **chars**$(\_, chs)) = (E.typedec)[ktypeno]$ **in**
  let $index = \iota(i \in$ **inds** $chs) \cdot chs[i] = char$ **in**
    enum$(ktypeno,\ index)$
pre
  $Find\text{-}Nm(tagname, E) \in$ **consttag**

$Find\text{-}Assoc: Altname \times Env \rightarrow kTypeOpt$

$Find\text{-}Assoc(altname, E) \triangleq$
  let **consttag**$(ktypeno) = Find\text{-}Nm(altname, E)$ **in**
  let **typedec**$(\_,$ **tags**$(tags)) = (E.typedec)[ktypeno]$ **in**
   $\iota(ktypeOpt \in kTypeOpt) \cdot$ **tag**$(altname, ktypeOpt) \in$ **elems** $tags$
pre
  $Find\text{-}Nm(altname, E) \in$ **consttag**

$Find\text{-}Signal: Signal \times Env \rightarrow kUnit \times kType$

$Find\text{-}Signal(signalname, E) \triangleq$
  let **sig**$(signalno, ktype, \_) = Find\text{-}Nm(signalname, E)$ **in**
    signal$(signalno),\ ktype$
pre
  $Find\text{-}Nm(signalname, E) \in$ **sig**

4.4.8 Type Checking

Type checking is an important aspect of the ELLA compiler and this relation shows how the transformation from Core ELLA to the **Kernel** will define type equality.

$\boxed{\text{TE1}}$	$\textbf{typeno}(typeno_1) =\!\boxed{T}\!= \textbf{typeno}(typeno_2)$	$\Leftrightarrow$	$(typeno_1 = typeno_2)$
$\boxed{\text{TE2}}$	$\textbf{stringtype}(s_1, tn_1) =\!\boxed{T}\!= \textbf{stringtype}(s_2, tn_2)$	$\Leftrightarrow$	$(s_1 = s_2) \wedge (tn_1 = tn_2)$
$\boxed{\text{TE3}}$	$\textbf{types}[t_1, \ldots, t_k] =\!\boxed{T}\!= \textbf{types}[s_1, \ldots, s_k]$	$\Leftrightarrow$	$\bigwedge_{i=1..k} (t_i =\!\boxed{T}\!= s_i)$
$\boxed{\text{TE4}}$	$\textbf{typename}(\_, type_1) =\!\boxed{T}\!= type_2$	$\Leftrightarrow$	$(type_1 =\!\boxed{T}\!= type_2)$
$\boxed{\text{TE5}}$	$type_1 =\!\boxed{T}\!= \textbf{typename}(\_, type_2)$	$\Leftrightarrow$	$(type_1 =\!\boxed{T}\!= type_2)$

4.4.9 Character Check

This rule checks that a particular type is of the form of an ELLA character.

$Is\text{-}Char : \mathbb{N}_1 \times Env \rightarrow \mathbb{B}$

$Is\text{-}Char(typeno, E) \quad \triangleq \quad (E.typedec)[typeno].new \in$ **chars**

4.4.10 Type Indexing

This function describes how to obtain the type of an indexed type

$Index : kType \times \mathbb{N}_1 \rightarrow kType$

$Index(ty, i) \quad \triangleq$
cases ty **of**
types$[ktype_1, \cdots, ktype_k] \rightarrow ktype_i,$
typename$(\_, ktype) \rightarrow Index(ktype, i),$
stringtype$(\_, typeno) \rightarrow$ **typeno**$(typeno)$
end

4.4.11 Two Value Types

Here we present the function for checking that a type is a two valued enumerated type:

$Check\text{-}Two\text{-}Val : kType \rightarrow \mathbb{B}$

$Check\text{-}Two\text{-}Val($ **typeno**$(typeno)) \quad \triangleq \quad$ **len** ((E.typedec)[typeno])= 2

4.4.12 Reform

This function flattens types so that they are available for **reform**

$Flatten : kType \rightarrow kTypeSeq$

$Flatten(ty) \quad \triangleq$
cases ty **of**
typeno $(typeno) \rightarrow$ **typeno**$(typeno),$
stringtype $(size, typeno) \rightarrow$ **stringtype**$(size, typeno),$
typename $(\_, type) \rightarrow Flatten(type),$
types $[t_1, \cdots, t_k] \rightarrow Flatten(t_1) \frown \cdots \frown Flatten(t_k)$
end

4.4.13 Constructing Tuples

These functions convert sequences into tuples.

$TypeTuple : kTypeSeq \rightarrow kType$

$TypeTuple(tseq) \quad \triangleq$ **if len** $tseq = 1$
then $tseq[1]$
else types $tseq$

$ConstTuple : kConstSeq \rightarrow kConst$

$$ConstTuple(cseq) \quad \triangleq \quad \begin{array}{l} \textbf{if len}\ cseq = 1 \\ \textbf{then}\ cseq[1] \\ \textbf{else}\ \ \textbf{consts}\ cseq \end{array}$$

$ConstsetTuple : kConstsetSeq \rightarrow kConstset$

$$ConstsetTuple(csseq) \quad \triangleq \quad \begin{array}{l} \textbf{if len}\ csseq = 1 \\ \textbf{then}\ csseq[1] \\ \textbf{else}\ \ \textbf{constsets}\ csseq \end{array}$$

$UnitTuple : kUnitSeq \rightarrow kUnit$

$$UnitTuple(useq) \quad \triangleq \quad \begin{array}{l} \textbf{if len}\ useq = 1 \\ \textbf{then}\ useq[1] \\ \textbf{else}\ \ \textbf{units}\ useq \end{array}$$

4.4.14 Case Disjointness

Within CASE statements all the different arms must have distinctive choosers. In order to ensure this the following rules are needed.

$\boxed{\text{CD1}}$ $\quad \textbf{enum}(\_, tagno_1)\ Disjoint\ \ \textbf{enum}(\_, tagno_2) \ \Leftrightarrow\ tagno_1 \neq tagno_2$

$\boxed{\text{CD2}}$ $\quad \textbf{string}(\_, [tagno_{11}, \cdots, tagno_{1k}])$
$\quad Disjoint$
$\quad \quad \textbf{string}(\_, [tagno_{21}, \cdots, tagno_{2k}])$
$$\Leftrightarrow \bigvee_{i=\{1..k\}} (tagno_{1i} \neq tagno_{2i})$$

$\boxed{\text{CD3}}$ $\quad \textbf{constsetassoc}(\ \textbf{enum}(\_, tagno_1), constset_1)$
$\quad Disjoint$
$\quad \quad \textbf{constsetassoc}(\ \textbf{enum}(\_, tagno_2), constset_2)$
$$\Leftrightarrow\ tagno_1 \neq tagno_2 \vee constset_1\ Disjoint\ constset_2$$

$\boxed{\text{CD4}}$ $\quad \textbf{constsets}[csa_1, \cdots, csa_k]\ Disjoint\ \ \textbf{constsets}[csb_1, \cdots, csb_k] \ \Leftrightarrow$
$$\bigvee_{i=\{1..k\}} (csa_i\ Disjoint\ csb_i)$$

$\boxed{\text{CD5}}$ $\quad enum\ Disjoint\ \mathbf{constsetalts}[csa_1, \cdots, csa_k] \Leftrightarrow \bigwedge_{i=\{1..k\}} (enum\ Disjoint\ csa_i)$

$\boxed{\text{CD6}}$ $\quad constsetalts\ Disjoint\ enum \Leftrightarrow enum\ Disjoint\ constsetalts$

$\boxed{\text{CD7}}$ $\quad \mathbf{constsetalts}[csa_1, \cdots, csa_k]\ Disjoint\ constsetalts \Leftrightarrow \bigwedge_{i=\{1..k\}} (csa_i\ Disjoint\ constsetalts)$

4.4.15 Concatenation

Concatenation of two signal types are handled by means of the following rules.

$\boxed{\text{CAT1}}$ $\quad \mathbf{typename}(\_, type_1)\ Conc\ type_2 = type_1\ Conc\ type_2$

$\boxed{\text{CAT2}}$ $\quad type_1\ Conc\ \mathbf{typename}(\_, type_2) = type_1\ Conc\ type_2$

$\boxed{\text{CAT3}}$ $\quad \forall i \in \{1..k\}, j \in \{1..l\} \cdot (ta_i = \boxed{\text{T}} = tb_j) \Rightarrow$
$\mathbf{types}[ta_1, \cdots, ta_k]\ Conc\ \mathbf{types}[tb_1, \cdots, tb_l] = \mathbf{types}[ta_1, \cdots, ta_k, tb_1, \cdots, tb_l]$

$\boxed{\text{CAT4}}$ $\quad \forall i \in \{1..k\} \cdot t_i = \boxed{\text{T}} = type \Rightarrow$
$\mathbf{types}[t_1, \cdots, t_k]\ Conc\ type = \mathbf{types}[t_1, \cdots, t_k, type]$

$\boxed{\text{CAT5}}$ $\quad \forall i \in \{1..k\} \cdot t_i = \boxed{\text{T}} = type \Rightarrow$
$type\ Conc\ \mathbf{types}[t_1, \cdots, t_k] = \mathbf{types}[type, t_1, \cdots, t_k]$

4.5 Transformations from Core ELLA to the Kernel

This section describes transformations from Core ELLA to the **Kernel**. These include the semantic checks which are done by the full ELLA compiler on Core ELLA ie. type checking, name checking etc. Thus this section includes a description of the static semantics of Core ELLA. At the start of each subsection the Core ELLA syntax, for which the transformations of that section apply, will be given.

4.5.1 Enumerated

Enumerated values are defined by

enumerated ::= altname
| tagname / z
| tagname 'char
| tagname "string"

and the transformations on them are given by

$\boxed{\text{EM1}}$
$$E \vdash [\![altname]\!] = \boxed{\text{EM}} \Rightarrow Find\text{-}Alt\,(altname, E)$$

$$\boxed{\text{EM2}}\ \frac{\textit{Find-ELLAint}\,(tagname, E) \ = ktypeno,\ lb,\ ub \qquad lb \leq z \leq ub}{E \vdash [\![tagname/z]\!] \ \boxed{\text{EM}}\!\!\Rightarrow\ \mathbf{enum}(ktypeno, z\text{-}lb+1)}$$

$$\boxed{\text{EM3}}\ \frac{}{E \vdash [\![tagname\,'char]\!] \ \boxed{\text{EM}}\!\!\Rightarrow\ \textit{Find-Char}\,(tagname, char, E)}$$

$$\boxed{\text{EM4}}\ \frac{\textit{Find-Char}\,(tagname, char_i, E) \ = \ \mathbf{enum}(ktypeno, ktagno_i) \qquad \textit{ForAll}\ i \in \{1..k\}}{E \vdash [\![tagname\,''char_1 \cdots char_k'']\!] \ \boxed{\text{EM}}\!\!\Rightarrow\ \mathbf{string}(ktypeno, [ktagno_1, \cdots, ktagno_k])}$$

4.5.2 Type

Types in Core ELLA can have the following form

type	::=	typename
	|	STRING [size] type
	|	[size] type
	|	($\text{type}_1, \cdots, \text{type}_k$)

and the transformations that apply to them are

$$\boxed{\text{T1}}\ \frac{}{E \vdash [\![typename]\!] \ \boxed{\text{T}}\!\!\Rightarrow\ \textit{Find-Type}\,(typename, E)}$$

$$\boxed{\text{T2}}\ \frac{E \vdash [\![type]\!] \ \boxed{\text{T}}\!\!\Rightarrow\ \mathbf{typeno}(ktypeno) \qquad \textit{Is-Char}\,(ktypeno)}{E \vdash [\![STRING[size]type]\!] \ \boxed{\text{T}}\!\!\Rightarrow\ \mathbf{stringtype}(size, ktypeno)}$$

$$\boxed{\text{T3}}\ \frac{E \vdash [\![type]\!] \ \boxed{\text{T}}\!\!\Rightarrow\ ktype}{E \vdash [\![[size]type]\!] \ \boxed{\text{T}}\!\!\Rightarrow\ \mathbf{types}[ktype^{size}]}$$

$$\boxed{\text{T4}}\ \frac{E \vdash [\![type_i]\!] \ \boxed{\text{T}}\!\!\Rightarrow\ t_i \qquad \textit{ForAll}\ i \in \{1..k\}}{E \vdash [\![(type_1, \cdots, type_k)]\!] \ \boxed{\text{T}}\!\!\Rightarrow\ \textit{TypeTuple}([t_1, \cdots, t_k])}$$

4.5.3 Constant

The Core ELLA definition of constants is

const	::=	STRING [size] const1
	|	[size] const
	|	const1

const1 ::= enumerated
| tagname & const1
| ($\text{const}_1, \cdots, \text{const}_k$)
| ? type

with their transformation rules being

$$\text{C1}\ \frac{E \vdash [\![const]\!] \boxed{\text{C}} \Rightarrow kconst\colon \mathbf{typeno}(ktypeno) \qquad \textit{Is-Char}\,(ktypeno,\ E)}{E \vdash [\![STRING[size]const]\!] \boxed{\text{C}} \Rightarrow \mathbf{conststring}(size, kconst)\colon \mathbf{stringtype}(size, ktypeno)}$$

$$\text{C2}\ \frac{E \vdash [\![const]\!] \boxed{\text{C}} \Rightarrow kconst\colon ktype}{E \vdash [\![[size]const]\!] \boxed{\text{C}} \Rightarrow \mathbf{consts}[kconst^{size}]\colon \mathbf{types}[ktype^{size}]}$$

$$\text{C3}\ \frac{E \vdash [\![enumerated]\!] \boxed{\text{EM}} \Rightarrow \mathbf{enum}(ktypeno, tagno)}{E \vdash [\![enumerated]\!] \boxed{\text{C}} \Rightarrow \mathbf{enum}(ktypeno, tagno)\colon \mathbf{typeno}(ktypeno)}$$

$$\text{C4}\ \frac{E \vdash [\![const]\!] \boxed{\text{C}} \Rightarrow kconst\colon ktype_1 \qquad \textit{Find-Assoc}\,(altname, E) = ktype_2 \qquad ktype_1 \boxed{\text{T}} ktype_2 \qquad \textit{Find-Alt}\,(altname, E) = \mathbf{enum}(ktypeno, \_)}{E \vdash [\![altname\&const]\!] \boxed{\text{C}} \Rightarrow \mathbf{constassoc}(\textit{Find-Alt}\,(altname, E), kconst)\colon \mathbf{typeno}(ktypeno)}$$

$$\text{C5}\ \frac{E \vdash [\![const_i]\!] \boxed{\text{C}} \Rightarrow kconst_i\colon ktype_i \qquad \textit{ForAll}\ i \in \{1..k\}}{E \vdash [\![(const_1, \cdots, const_k)]\!] \boxed{\text{C}} \Rightarrow \textit{ConstTuple}([kconst_1, \cdots, kconst_k])\colon \textit{TypeTuple}([ktype_1, \cdots, ktype_k])}$$

$$\text{C6}\ \frac{E \vdash [\![type]\!] \boxed{\text{T}} \Rightarrow ktype}{E \vdash [\![?type]\!] \boxed{\text{C}} \Rightarrow \mathbf{constquery}(ktype)\colon ktype}$$

4.5.4 Constset

Constant sets are given by

constset ::= $\text{constset}_1 \mid \cdots \mid \text{constset}_k$

constset1 ::= STRING [size] constset2
| [size] constset1
| constset2

constset2 ::= enumerated
| tagname & constset2
| ($\text{constset}_1, \cdots, \text{constset}_k$)
| type

with transformations on them given by

$$\boxed{\text{CS1}}\ \frac{E \vdash [\![constset_i]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow kcset_i : ktype_i \qquad ktype_i \mathrel{=}\!\boxed{\text{T}}\!= ktype_1 \qquad ForAll\ i \in \{1..k\}}{E \vdash [\![constset_1 \mid \cdots \mid constset_k]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow \ \mathbf{constsetalts}[kcset_1, \cdots, kcset_k] : ktype_1}$$

$$\boxed{\text{CS2}}\ \frac{E \vdash [\![constset]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow kcset:\ \mathbf{typeno}(ktypeno) \qquad Is\text{-}Char\ (ktypeno,\ E)}{E \vdash [\![STRING[size]constset]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow \mathbf{constsetstring}(size, kcset):\ \mathbf{stringtype}(size, ktypeno)}$$

$$\boxed{\text{CS3}}\ \frac{E \vdash [\![constset]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow kcset : ktype}{E \vdash [\![[size]constset]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow \ \mathbf{constsets}[kcset^{size}]:\ \mathbf{types}[ktype^{size}]}$$

$$\boxed{\text{CS4}}\ \frac{E \vdash [\![enumerated]\!] \mathrel{=}\!\boxed{\text{EM}}\!\Rightarrow \ \mathbf{enum}(ktypeno, tagno)}{E \vdash [\![enumerated]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow \ \mathbf{enum}(ktypeno, tagno):\ \mathbf{typeno}(ktypeno)}$$

$$\boxed{\text{CS5}}\ \frac{\begin{array}{c} E \vdash [\![constset]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow kcset : ktype_1 \qquad Find\text{-}Assoc\ (altname, E)\ = ktype_2 \\ ktype_1 \mathrel{=}\!\boxed{\text{T}}\!= ktype_2 \qquad Find\text{-}Alt\ (altname, E)\ =\ \mathbf{enum}(typeno, \_) \end{array}}{\begin{array}{c} E \vdash [\![altname\&constset]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow \\ \mathbf{constsetassoc}(Find\text{-}Alt\ (altname, E)\ , kcset):\ \mathbf{typeno}(typeno) \end{array}}$$

$$\boxed{\text{CS6}}\ \frac{E \vdash [\![constset_i]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow kcset_i : ktype_i \qquad Forall\ i \in \{1..k\}}{\begin{array}{c} E \vdash [\![(constset_1, \cdots, constset_k)]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow \\ ConstsetTuple([kcset_1, \cdots, kcset_k]) : TypeTuple([ktype_1, \cdots, ktype_k]) \end{array}}$$

$$\boxed{\text{CS7}}\ \frac{E \vdash [\![type]\!] \mathrel{=}\!\boxed{\text{T}}\!\Rightarrow ktype}{E \vdash [\![type]\!] \mathrel{=}\!\boxed{\text{CS}}\!\Rightarrow \ \mathbf{constsetany}(ktype) : ktype}$$

4.5.5 Units

The complete Core ELLA unit syntax is given by

unit	::=	unit CONC unit1
		| unit1
unit1	::=	STRING [size] unit1
		| [size] unit1
		| fnname unit1
		| tagname & unit1
		| unit2 // tagname
		| unit2
unit2	::=	signalname
		| enumerated
		| unit2 [index]
		| ? type
		| closedclause

with the transformations defined by

$$\boxed{\text{U1}}\quad \frac{}{E \vdash [\![\mathit{signalname}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{Find\text{-}Signal}\,(\mathit{signalname}, E),\ E}$$

$$\boxed{\text{U2}}\quad \frac{E \vdash [\![\mathit{enumerated}]\!] \dashv\!\boxed{\text{EM}}\!\Rightarrow \ \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno})}{E \vdash [\![\mathit{enumerated}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \ \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno}):\ \mathbf{typeno}(\mathit{ktypeno}),\ E}$$

$$\boxed{\text{U3}}\quad \frac{\begin{array}{c} E_1 \vdash [\![\mathit{unit}_1]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_1\colon\ \mathit{ktype}_1,\ E_2 \\ E_2 \vdash [\![\mathit{unit}_2]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_2\colon\ \mathit{ktype}_2,\ E_3 \\ \mathit{ktype}_{out} = \mathit{ktype}_1\, \mathit{Conc}\ \mathit{ktype}_2 \end{array}}{E \vdash [\![\mathit{unit}_1 \mathit{CONC}\ \mathit{unit}_2]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \ \mathbf{conc}(\mathit{kunit}_1, \mathit{kunit}_2, \mathit{ktype}_{out})\colon\ \mathit{ktype}_{out},\ E_3}$$

$$\boxed{\text{U4}}\quad \frac{\begin{array}{c} E \vdash [\![\mathit{unit}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}\colon\ \mathbf{typeno}(\mathit{ktypeno}),\ E' \\ \mathit{Is\text{-}Char}\,(\mathit{ktypeno},\ E) \end{array}}{\begin{array}{c} E \vdash [\![\mathit{STRING}[\mathit{size}]\mathit{unit}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \\ \mathbf{unitstring}(\mathit{size}, \mathit{kunit})\colon\ \mathbf{stringtype}(\mathit{size}, \mathit{ktypeno}),\ E' \end{array}}$$

$$\boxed{\text{U5}}\quad \frac{E \vdash [\![\mathit{unit}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}\colon\ \mathit{ktype},\ E'}{E \vdash [\![[\mathit{size}]\mathit{unit}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \ \mathbf{units}[\mathit{kunit}^{\mathit{size}}]\colon\ \mathbf{types}[\mathit{ktype}^{\mathit{size}}],\ E'}$$

$$\frac{\begin{array}{c} E \vdash [\![unit]\!] \boxed{\text{U}} \Rightarrow kunit\colon ktype,\ E' \\ fnno = \textit{Find-Fn}\,(fnname, E) \\ (E.fndec)[fnno].inputtype \boxed{\text{T}} = ktype \end{array}}{E \vdash [\![fnname\ unit]\!] \boxed{\text{U}} \Rightarrow\ \mathbf{call}(fnno, kunit)\colon ((E.fndec)[fnno].outputtype),\ E'}\ \boxed{\text{U6}}$$

$$\frac{\begin{array}{c} E \vdash [\![unit_1]\!] \boxed{\text{U}} \Rightarrow kunit_1\colon ktype_1,\ E' \\ \textit{Find-Assoc}\,(altname, E)\ = ktype_2 \\ ktype_1 \boxed{\text{T}} = ktype_2 \qquad \textit{Find-Alt}\,(altname, E)\ =\ \mathbf{enum}(typeno, \_) \end{array}}{\begin{array}{c} E \vdash [\![altname\&unit_1]\!] \boxed{\text{U}} \Rightarrow \\ \mathbf{unitassoc}(\textit{Find-Alt}\,(altname, E), kunit_1)\colon\ \mathbf{typeno}(typeno),\ E' \end{array}}\ \boxed{\text{U7}}$$

$$\frac{\begin{array}{c} E \vdash [\![unit]\!] \boxed{\text{U}} \Rightarrow kunit\colon ktype,\ E' \qquad \textit{Find-Assoc}\,(altname, E)\ = ktype, \\ \textit{Find-Alt}\,(altname, E)\ =\ \mathbf{enum}(typeno, \_), \\ \mathbf{typeno}(typeno) \boxed{\text{T}} = ktype \end{array}}{E \vdash [\![unit//altname]\!] \boxed{\text{U}} \Rightarrow\ \mathbf{restrict}(kunit, \textit{Find-Alt}\,(altname, E), t)\colon ktype,\ E'}\ \boxed{\text{U8}}$$

$$\frac{E \vdash [\![unit]\!] \boxed{\text{U}} \Rightarrow kunit\colon ktype,\ E' \qquad Index(ktype, index) = t}{E \vdash [\![unit[index]]\!] \boxed{\text{U}} \Rightarrow\ \mathbf{index}(kunit, index, t)\colon t,\ E'}\ \boxed{\text{U9}}$$

$$\frac{E \vdash [\![type]\!] \boxed{\text{T}} \Rightarrow ktype}{E \vdash [\![?type]\!] \boxed{\text{U}} \Rightarrow\ \mathbf{unitquery}(ktype)\colon ktype,\ E}\ \boxed{\text{U10}}$$

$$\frac{E \vdash [\![closedclause]\!] \boxed{\text{CC}} \Rightarrow kunit\colon ktype,\ E'}{E \vdash [\![closedclause]\!] \boxed{\text{U}} \Rightarrow kunit\colon ktype,\ E'}\ \boxed{\text{U11}}$$

4.5.6 Closedclause

Closed clauses are given by

closedclause	::=	CASE unit OF cases ELSE unit ESAC
	|	($\text{unit}_1, \cdots, \text{unit}_k$)
	|	BEGIN $\text{step}_1 \cdots \text{step}_{k\text{-}1}$ OUTPUT unit END
cases	::=	$\text{constset}_1 : \text{unit}_1, \cdots, \text{constset}_k : \text{unit}_k$
step	::=	LET $\text{letdecs}_1, \cdots, \text{letdecs}_k$.
	|	MAKE $\text{makedecs}_1, \cdots, \text{makedecs}_k$.
	|	JOIN $\text{joindecs}_1, \cdots, \text{joindecs}_k$.

letdecs	::=	signalname = unit
makedecs	::=	fnname : signalname
joindecs	::=	unit → signalname

with the transformations given by

$$\boxed{\mathrm{CA}}\ \frac{\begin{array}{c} E \vdash [\![constset]\!] =\!\boxed{\mathrm{CS}}\!\Rightarrow kconstset\colon ktypeconst \\ E \vdash [\![unit]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit\colon\ ktype,\ E' \end{array}}{E \vdash [\![constset\colon unit]\!] =\!\boxed{\mathrm{CA}}\!\Rightarrow kconstset,\ kunit\colon\ ktypeconst,\ ktype,\ E'}$$

$$\boxed{\mathrm{CC1}}\ \frac{\begin{array}{c} E \vdash [\![unit_1]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_{in}\colon\ ktype_{in},\ E_1 \\ E_i \vdash [\![case_i]\!] =\!\boxed{\mathrm{CA}}\!\Rightarrow kcs_i,\ ku_i\colon\ ktypec_i,\ ktypeu_i,\ E_{i+1} \qquad ForAll\ i \in \{1..k\} \\ E_{k+1} \vdash [\![unit_2]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_{out}\colon\ ktype_{out},\ E' \\ ktypec_i =\!\boxed{\mathrm{T}}\!= ktype_{in} \qquad ktypeu_i =\!\boxed{\mathrm{T}}\!= ktype_{out} \qquad ForAll\ i \in \{1..k\} \\ kcs_i\ Disjoint\ kcs_j \qquad ForAll\ i,j \in \{1..k\}\ i \neq j \end{array}}{\begin{array}{c} E \vdash [\![CASE\ unit_1 OF\ case_1, \cdots, case_k ELSE\ unit_2 ESAC]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow \\ \mathbf{caseclause}(kunit_{in}, [\ \mathbf{case}(kcs_1, ku_1), \cdots,\ \mathbf{case}(kcs_k, ku_k)], kunit_{out}, ktype_{out})\colon\ ktype_{out},\ E' \end{array}}$$

$$\boxed{\mathrm{CC2}}\ \frac{E_i \vdash [\![unit_i]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_i\colon\ ktype_i,\ E_{i+1} \qquad ForAll\ i \in \{1..k\}}{\begin{array}{c} E_1 \vdash [\![(unit_1, \cdots, unit_k)]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow \\ UnitTuple([kunit_1, \cdots, kunit_k])\colon\ TypeTuple([ktype_1, \cdots, ktype_k]),\ E_{k+1} \end{array}}$$

$$\boxed{\mathrm{CC3}}\ \frac{\begin{array}{c} Scope\text{-}Begin\ (E)\ = E_1 \qquad Check\text{-}Joins\ (E_{k+1}) \qquad Scope\text{-}End\ (E, E_{k+1}) = E' \\ E_i \vdash [\![step_i]\!] =\!\boxed{\mathrm{SP}}\!\Rightarrow E_{i+1} \qquad ForAll\ i \in \{1..k\text{-}1\} \\ E_k \vdash [\![unit]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit\colon\ ktype,\ E_{k+1} \end{array}}{E \vdash [\![BEGIN\ step_1 \cdots step_{k\text{-}1} OUTPUT\ unit\ END]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow kunit\colon\ ktype,\ E'}$$

$$\boxed{\mathrm{SP1}}\ \frac{E \vdash [\![unit]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit\colon\ ktype,\ E'}{\begin{array}{c} E \vdash [\![LET\ signalname = unit]\!] =\!\boxed{\mathrm{SP}}\!\Rightarrow \\ Add\text{-}Signal\ (\ \mathbf{signaldec}(signalname, ktype, kunit),\ \mathbf{joined}, E') \end{array}}$$

$$\boxed{\mathrm{SP2}}\ \frac{E_i \vdash [\![LET\ signalname_i = unit_i]\!] =\!\boxed{\mathrm{SP}}\!\Rightarrow E_{i+1} \qquad ForAll\ i \in \{1..k\}}{E_1 \vdash [\![LET\ signalname_1 = unit_1, \cdots, signalname_k = unit_k]\!] =\!\boxed{\mathrm{SP}}\!\Rightarrow E_{k+1}}$$

$$\boxed{\mathrm{SP3}}\ \frac{Find\text{-}Fn\ (fnname, E)\ = fnno}{\begin{array}{c} E \vdash [\![MAKE\ fnname\colon signalname]\!] =\!\boxed{\mathrm{SP}}\!\Rightarrow \\ Add\text{-}Signal\ (\ \mathbf{signaldec}(signalname, ((E.fndec)[fnno].outputtype),\ \mathbf{call}(fnno, \_)),\ \mathbf{unjoined}, E) \end{array}}$$

$$\boxed{\text{SP4}}\ \frac{E_i \vdash [\![\mathit{MAKE\ fnname_i}\colon \mathit{signalname_i}]\!] \Rightarrow\!\boxed{\text{SP}}\!\Rightarrow E_{i+1} \qquad \mathit{ForAll}\ i \in \{1..k\}}{E_1 \vdash [\![\mathit{MAKE\ fnname_1}\colon \mathit{signalname_1}, \cdots, \mathit{fnname_k}\colon \mathit{signalname_k}]\!] \Rightarrow\!\boxed{\text{SP}}\!\Rightarrow E_{k+1}}$$

$$\boxed{\text{SP5}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{unit}]\!] \Rightarrow\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}\colon \mathit{ktype},\ E' \\ \mathit{Find\text{-}Unjoined}\,(\mathit{signalname}, E)\ = \mathit{fnno} \\ \mathit{Find\text{-}Signal}\,(\mathit{signalname}, E)\ =\ \mathbf{signal}(\mathit{signalno})\colon\_ \\ (E.\mathit{fndec})[\mathit{fnno}].\mathit{inputtype} =\!\boxed{\text{T}}\!= \mathit{ktype} \end{array}}{\begin{array}{c} E \vdash [\![\mathit{JOIN\ unit} \longrightarrow\ \mathit{signalname}]\!] \Rightarrow\!\boxed{\text{SP}}\!\Rightarrow \\ \mathit{Add\text{-}Join}\ (\ \mathbf{signaldec}(\mathit{signalname}, \mathit{ktype},\ \mathbf{call}(\mathit{fnno}, \mathit{kunit})), \mathit{signalno}, E') \end{array}}$$

$$\boxed{\text{SP6}}\ \frac{E_i \vdash [\![\mathit{JOIN\ unit_i} \longrightarrow\ \mathit{signalname_i}]\!] \Rightarrow\!\boxed{\text{U}}\!\Rightarrow E_{i+1} \qquad \mathit{ForAll}\ i \in \{1..k\}}{E_1 \vdash [\![\mathit{JOIN\ unit_1} \longrightarrow\ \mathit{signalname_1}, \cdots, \mathit{unit_k} \longrightarrow\ \mathit{signalname_k}]\!] \Rightarrow\!\boxed{\text{SP}}\!\Rightarrow E_{k+1}}$$

4.5.7 Built-In Functions

Built-in-functions (function bodies) are defined to be

functionbody ::= unit
| REFORM
| BIOP biopname
| DELAY (initial_value, ambig_time, ambig_value, delay_time)
| IDELAY (initial_value, delay_time)
| SAMPLE (interval, initial_value, skew_time)
| RAM (initial_value)

with the following transformations

$$\boxed{\text{BI1}}\ \frac{\mathit{TypeTuple}(\mathit{Flatten}(\mathit{ktype_{in}})) =\!\boxed{\text{T}}\!= \mathit{TypeTuple}(\mathit{Flatten}(\mathit{ktype_{out}}))}{E \vdash [\![\mathit{REFORM}]\!]\ \{\mathit{ktype_{in}}, \mathit{ktype_{out}}\} \Rightarrow\!\boxed{\text{BI}}\!\Rightarrow\ \mathbf{reform}}$$

$$\boxed{\text{BI2}}\ \frac{\mathit{Type\ Checking\ to\ be\ added}}{E \vdash [\![\mathit{BIOP\ biopname}]\!]\ \{\mathit{ktype_{in}}, \mathit{ktype_{out}}\} \Rightarrow\!\boxed{\text{BI}}\!\Rightarrow\ \mathbf{biop}(\mathit{biopname})}$$

$$\boxed{\text{BI3}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{initial\_value}]\!] \Rightarrow\!\boxed{\text{C}}\!\Rightarrow \mathit{kconst_i}\colon \mathit{ktype_i} \\ E \vdash [\![\mathit{ambig\_value}]\!] \Rightarrow\!\boxed{\text{C}}\!\Rightarrow \mathit{kconst_a}\colon \mathit{ktype_a} \\ \mathit{ktype_{in}} =\!\boxed{\text{T}}\!= \mathit{ktype_{out}} =\!\boxed{\text{T}}\!= \mathit{ktype_i} =\!\boxed{\text{T}}\!= \mathit{ktype_a} \\ \mathit{ambig\_time} \geq\ \mathit{delay\_time} \end{array}}{\begin{array}{c} E \vdash [\![\mathit{DELAY}(\mathit{initial\_value}, \mathit{ambig\_time}, \mathit{ambig\_value}, \mathit{delay\_time})]\!]\ \{\mathit{ktype_{in}}, \mathit{ktype_{out}}\} \Rightarrow\!\boxed{\text{BI}}\!\Rightarrow \\ \mathbf{delay}(\mathit{kconst_i}, \mathit{ambig\_time}, \mathit{kconst_a}, \mathit{delay\_time}) \end{array}}$$

$$\boxed{\text{BI4}}\;\frac{E \vdash [\![\mathit{initial\_value}]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow \mathit{kconst}\!:\!\mathit{ktype} \qquad \mathit{ktype} =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_{in} =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_{out}}{\begin{array}{c}E \vdash [\![\mathit{IDELAY}(\mathit{initial\_value}, \mathit{delay\_time})]\!]\ \{\mathit{ktype}_{in}, \mathit{ktype}_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow \\ \mathbf{idelay}(\mathit{kconst}, \mathit{delay\_time})\end{array}}$$

$$\boxed{\text{BI5}}\;\frac{E \vdash [\![\mathit{initial\_value}]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow \mathit{kconst}\!:\!\mathit{ktype}}{\begin{array}{c}E \vdash [\![\mathit{SAMPLE}(\mathit{interval}, \mathit{initial\_value}, \mathit{skew})]\!]\ \{\mathit{ktype}_{in}, \mathit{ktype}_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow \\ \mathbf{sample}(\mathit{interval}, \mathit{kconst}, \mathit{skew})\end{array}}$$

$$\boxed{\text{BI6}}\;\frac{\begin{array}{c}E \vdash [\![\mathit{initial\_value}]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow \mathit{kconst}_I\!:\!\mathit{ktype}_I \\ \mathit{ktype}_{in} = \mathbf{types}(\mathit{ktype}_{data}, \mathit{ktype}_{writeaddress}, \mathit{ktype}_{readaddress}, \mathit{ktype}_{writeenable}) \\ \mathit{ktype}_{data} =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_{out} =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_I \\ \mathit{Find\text{-}Integer\text{-}Type}\ (\mathit{ktype}_{writeaddress}) = \_\,,\ \mathit{lb},\ \mathit{ub} \\ \mathit{Find\text{-}Integer\text{-}Type}\ (\mathit{ktype}_{readaddress}) = \_\,,\ \mathit{lb},\ \mathit{ub} \qquad \mathit{lb} = 1 \\ \mathit{Check\text{-}Two\text{-}Val}\ (\mathit{ktype}_{writeenable})\end{array}}{E \vdash [\![\mathit{RAM}(\mathit{initial\_value})]\!]\ \{\mathit{ktype}_{in}, \mathit{ktype}_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow\ \mathbf{ram}(\mathit{kconst}_I)}$$

4.5.8 Type Declaration

Type declarations are defined as

typedec	::=	TYPE typename = typeornew.
typeornew	::=	type
	|	new
new	::=	NEW tagname / (lwb .. upb)
	|	NEW (typealt_1 | ⋯ | typealt_k)
	|	NEW tagname ($'\text{char}_1$ | ⋯ | $'\text{char}_k$)
typealt	::=	tagname & type
	|	altname

with transformations on them by

$$\boxed{\text{TD1}}\;\frac{E \vdash [\![\mathit{new}]\!] =\!\boxed{\mathrm{NW}}\!\Rightarrow \mathit{knew},\ E'}{E \vdash [\![\mathit{TYPE\ typename} = \mathit{new}.\ ]\!] =\!\boxed{\mathrm{TD}}\!\Rightarrow \mathit{Add\text{-}Type}\ (\ \mathbf{typedec}(\mathit{typename}, \mathit{knew}),\ E'\,)}$$

$$\boxed{\text{TD2}}\;\frac{E \vdash [\![\mathit{type}]\!] =\!\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}}{E \vdash [\![\mathit{TYPE\ typename} = \mathit{type}.\ ]\!] =\!\boxed{\mathrm{TD}}\!\Rightarrow \mathit{Add\text{-}Type\text{-}Name}\ (\mathit{typename}, \mathit{ktype}, E)}$$

$$\boxed{\text{NW1}}\;\frac{}{\begin{array}{c}E \vdash [\![\mathit{NEW\ tagname}/(\mathit{lwb}..\mathit{upb})]\!] =\!\boxed{\mathrm{NW}}\!\Rightarrow \\ \mathbf{ellaint}(\mathit{tagname}, \mathit{lwb}, \mathit{upb}),\ \mathit{Add\text{-}Tag}\ (\mathit{tagname}, E)\end{array}}$$

$$\boxed{\text{NW2}}\ \frac{E_i \vdash [\![typealt_i]\!] \boxed{\text{TA}}\Rightarrow t_i,\ E_{i+1} \qquad ForAll\ i \in \{1..k\}}{E_1 \vdash [\![NEW\ (typealt_1 \mid \cdots \mid typealt_k)]\!] \boxed{\text{NW}}\Rightarrow\ \textbf{tags}[t_1, \cdots, t_k],\ E_{k+1}}$$

$$\boxed{\text{NW3}}\ \frac{}{\begin{array}{c} E \vdash [\![NEW\ tagname('char_1 \mid \cdots \mid\ 'char_k)]\!] \boxed{\text{NW}}\Rightarrow \\ \textbf{chars}(tagname, [char_1, \cdots, char_k]),\ Add\text{-}Tag\ (tagname, E) \end{array}}$$

$$\boxed{\text{TA1}}\ \frac{E \vdash [\![type]\!] \boxed{\text{T}}\Rightarrow ktype}{E \vdash [\![tagname\&type]\!] \boxed{\text{TA}}\Rightarrow\ \textbf{tag}(tagname, ktype),\ Add\text{-}Tag\ (tagname, E)}$$

$$\boxed{\text{TA2}}\ \frac{}{E \vdash [\![altname]\!] \boxed{\text{TA}}\Rightarrow\ \textbf{tag}(altname, \{\textbf{nil}\}),\ Add\text{-}Tag\ (altname, E)}$$

4.5.9 Function Declaration

Function declarations are given by

fndec ::= FN fnname = (input) $\rightarrow$ type : functionbody.

input ::= type_1 : $\text{signalname}_1, \cdots, \text{type}_k$: signalname_k

and the transformations on them by

$$\boxed{\text{FD1}}\ \frac{\begin{array}{c} E \vdash [\![input]\!] \boxed{\text{IN}}\Rightarrow ktype_{inputs},\ E' \\ E' \vdash [\![type]\!] \boxed{\text{T}}\Rightarrow ktype_{out} \\ E' \vdash [\![unit]\!] \boxed{\text{U}}\Rightarrow kunit\colon ktype,\ E'' \\ ktype_{out} \boxed{\text{T}}= ktype \end{array}}{\begin{array}{c} E \vdash [\![FN\ fnname = (input) \longrightarrow type\colon unit.\ ]\!] \boxed{\text{FD}}\Rightarrow \\ Add\text{-}Fn\ (\ \textbf{fndec}(fnname, ktype_{inputs},\ E''.sigdec, ktype_{out}, kunit), E) \end{array}}$$

$$\boxed{\text{FD2}}\ \frac{\begin{array}{c} E \vdash [\![input]\!] \boxed{\text{IN}}\Rightarrow ktype_{inputs},\ E' \\ E' \vdash [\![type]\!] \boxed{\text{T}}\Rightarrow ktype_{out} \\ E' \vdash [\![builtin]\!]\ \{ktype_{inputs}, ktype\} \boxed{\text{BI}}\Rightarrow kbuiltin \end{array}}{\begin{array}{c} E \vdash [\![FN\ fnname = (input) \longrightarrow type\colon builtin.\ ]\!] \boxed{\text{FD}}\Rightarrow \\ Add\text{-}Fn\ (\ \textbf{fndec}(fnname, ktype_{inputs}, [\,], ktype_{out}, kbuiltin),\ E) \end{array}}$$

$$\boxed{\text{IN}}\ \frac{\begin{array}{c} E_1 \vdash [\![type_i]\!] \boxed{\text{T}}\Rightarrow ktype_i \qquad ForAll\ i \in \{1..k\} \\ Add\text{-}Signal(\ \textbf{signaldec}(signalname_i, ktype_i,\ \textbf{input}),\ \textbf{joined}, E_i) = E_{i+1} \qquad ForAll\ i \in \{1..k\} \end{array}}{\begin{array}{c} E_1 \vdash [\![type_1\colon signalname_1, \cdots, type_k\colon signalname_k]\!] \boxed{\text{IN}}\Rightarrow \\ TypeTuple([ktype_1, \cdots, ktypes_k]),\ E_{k+1} \end{array}}$$

4.5.10 Closure

A Closure is defined to be

closure ::= $\text{typedec}_1 \cdots \text{typedec}_j\ \text{fndec}_1 \cdots \text{fndec}_k$

with the following transforms

$$\boxed{\mathsf{CL}}\ \frac{\begin{array}{ll} E_i \vdash [\![typedec_i]\!] \models\!\boxed{\text{TD}}\!\Rightarrow E_{i+1} & ForAll\ i \in \{1..j\} \\ E_{l+j} \vdash [\![fndec_l]\!] \models\!\boxed{\text{FD}}\!\Rightarrow E_{l+1+j} & ForAll\ l \in \{1..k\} \end{array}}{E \vdash [\![typedec_1 \cdots typedec_j\ fndec_1 \cdots fndec_k]\!] \models\!\boxed{\text{CL}}\!\Rightarrow E_{1+j+k}}$$

$$\boxed{\mathsf{KERNEL}}\ \frac{InitialEnv \vdash [\![closure]\!] \models\!\boxed{\text{CL}}\!\Rightarrow E}{KERNEL([\![closure]\!]) = (E.typedec, E.fndec)}$$

4.6 Extracting the Type of a Kernel Structure

This section gives an example of how the **Kernel** can be used to find information about a circuit description. The example chosen is that of finding the type of any object in the **Kernel** and for checking type equality. This is achieved by defining two functions for extracting the type of a constant and unit. A function to check type equality is also given.

4.6.1 Constant Type Value

This section defines the function Type-Of-Const for getting the Type of a constant or constant set expression

$Type\text{-}Of\text{-}Const : (Const \cup Constset) \rightarrow kType$

$Type\text{-}Of\text{-}Const(const) \quad \triangleq$

$$\begin{array}{ll}
\textbf{cases}\ const\ \textbf{of} & \\
\quad \textbf{enum}(typeno, _) & \rightarrow typeno \\
\quad \textbf{string}(typeno, [tag_1, \cdots, tag_n]) & \rightarrow \textbf{stringtype}(n, typeno) \\
\quad \textbf{conststring}(size, c) & \rightarrow \textbf{stringtype}(size, Type\text{-}Of\text{-}Const(c)) \\
\quad \textbf{constassoc}(\ \textbf{enum}(typeno, _), _) & \rightarrow typeno \\
\quad \textbf{consts}[c_1, \cdots, c_n] & \rightarrow \textbf{types}[Type\text{-}Of\text{-}Const\ (c_1), \\
 & \quad \cdots, Type\text{-}Of\text{-}Const\ (c_n)] \\
\quad \textbf{constquery}(type) & \rightarrow type \\
\quad \textbf{constsetstring}(size, c) & \rightarrow \textbf{stringtype}(size, Type\text{-}Of\text{-}Const(c)) \\
\quad \textbf{constsetassoc}(\ \textbf{enum}(typeno, _), _) & \rightarrow typeno \\
\quad \textbf{constsets}[c_1, \cdots, c_n] & \rightarrow \textbf{types}[Type\text{-}Of\text{-}Const\ (c_1), \\
 & \quad \cdots, Type\text{-}Of\text{-}Const\ (c_n)] \\
\quad \textbf{constsetany}(type) & \rightarrow type \\
\quad \textbf{constsetalts}[c, \cdots] & \rightarrow Type\text{-}Of\text{-}Const\ (c) \\
\textbf{end} &
\end{array}$$

4.6.2 Unit Type Value

This section defines the function Type-Of-Unit for getting the Type of a unit expression

$$\mathit{Type\text{-}Of\text{-}Unit} : \mathit{FndecSeq} \times \mathit{Unit} \rightarrow \mathit{kType}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{Type\text{-}Of\text{-}Unit}(f, \mathit{unit}) \quad \triangleq} \\
\multicolumn{2}{l}{\quad \textbf{cases}\ \mathit{unit}\ \textbf{of}} \\
\quad \mathit{enumerated} & \rightarrow \mathit{Type\text{-}Of\text{-}Const}\ (\mathit{enumerated}) \\
\quad \textbf{conc}(_, _, \mathit{type}) & \rightarrow \mathit{type} \\
\quad \textbf{call}(\mathit{fnno}, _) & \rightarrow f[\mathit{fnno}].\mathit{outputtype} \\
\quad \textbf{unitassoc}(\ \textbf{enum}(\mathit{typeno}, _), _) & \rightarrow\ \textbf{typeno}(\mathit{typeno}) \\
\quad \textbf{restrict}(_, _, \mathit{outputtype}) & \rightarrow \mathit{outputtype} \\
\quad \textbf{signal}(\mathit{signalno}) & \rightarrow (f[\textbf{len} f].\mathit{decs})[\mathit{signalno}].\mathit{type} \\
\quad \textbf{index}(_, _, \mathit{outputtype}) & \rightarrow \mathit{outputtype} \\
\quad \textbf{unitquery}(\mathit{type}) & \rightarrow \mathit{type} \\
\quad \textbf{unitstring}(\mathit{size}, u) & \rightarrow\ \textbf{stringtype}(\mathit{size}, \mathit{Type\text{-}Of\text{-}Unit}\ (f, u)) \\
\quad \textbf{caseclause}(_, _, _, \mathit{outputtype}) & \rightarrow \mathit{outputtype} \\
\quad \textbf{units}[u_1, \cdots, u_n] & \rightarrow\ \textbf{types}[\mathit{Type\text{-}Of\text{-}Unit}\ (f, u_1), \cdots, \mathit{Type\text{-}Of\text{-}Unit}\ (f, u_n)] \\
\multicolumn{2}{l}{\quad \textbf{end}}
\end{array}$$

4.6.3 Equals Type Check

This section defines the function Type-Equals for checking that two types are equal

$$\mathit{Type\text{-}Equals} : \mathit{kType} \times \mathit{kType} \rightarrow \mathbb{B}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{Type\text{-}Equals}(t_1, t_2) \quad \triangleq} \\
\multicolumn{2}{l}{\quad \textbf{cases}\ (t_1, t_2)\ \textbf{of}} \\
\quad (\ \textbf{typeno}(t_1),\ \textbf{typeno}(t_2)) & \rightarrow (t_1 = t_2) \\
\quad (\ \textbf{stringtype}(s_1, t_1),\ \textbf{stringtype}(s_2, t_2)) & \rightarrow (t_1 = t_2) \wedge (s_1 = s_2) \\
\quad (\ \textbf{types}[t_1, \cdots, t_n],\ \textbf{types}[s_1, \cdots s_m]) & \rightarrow n = m \wedge (\ \bigwedge_{i=1..n} \mathit{Type\text{-}Equals}\ (t_i, s_i)) \\
\quad (\ \textbf{typename}(_, t_1), t_2) & \rightarrow \mathit{Type\text{-}Equals}\ (t_1, t_2) \\
\quad (t1,\ \textbf{typename}(_, t_2)) & \rightarrow \mathit{Type\text{-}Equals}\ (t_1, t_2) \\
\multicolumn{2}{l}{\quad \textbf{others false}} \\
\multicolumn{2}{l}{\quad \textbf{end}}
\end{array}$$

4.7 A Transformation Example

In this section we give an example of the transformation rules applied to a simple Core ELLA description. Through this example the reader will see how the scopes and types are incorporated into the environment and how the information is used.

Thus consider the following Core ELLA description

```
TYPE bool = NEW (t | f).
```

```
FN NOR = (bool:in1, bool:in2) -> bool:
CASE (in1, in2) OF
  (f,t):f,
  (t,f):f,
  (t,t):f,
 ELSE t
ESAC.

FN DEL = (bool:in) -> bool: DELAY (t,1,t,1).
```

with the initial environment

$$E = \mathbf{env}([\,],[\,],[\,],\{\},\{\},\{\},\{\},\{\},\{\},\{\})$$

Then we start be applying the transformation rules necessary for the type declaration

TA2 : $E \vdash [\![t]\!] \boxed{\text{TA}}\Rightarrow \mathbf{tag}(t,\{\}), E_1$

where

E_1 = Add-Tag (t, E)

i.e.

E_1 = ([], [], [], {}, {}, {}, {t $\mapsto$ **consttag**(1)}, {}, {}, {})

then

TA2 : $E_1 \vdash [\![f]\!] \boxed{\text{TA}}\Rightarrow \mathbf{tag}(f,\{\}), E_2$

where

E_2 = Add-Tag (f, E_1)

i.e.

E_2 = ([], [], [], {}, {}, {}, {f $\mapsto$ **consttag**(1), t $\mapsto$ **consttag**(1)}, {}, {}, {})

then

NW2: $E \vdash [\![NEW(t|f)]\!] \boxed{\text{NW}}\Rightarrow \mathbf{tags}[\,\mathbf{tag}(t,\{\}),\ \mathbf{tag}(f,\{\})], E_2$

which gives

TD1: $E \vdash [\![TYPE\ bool = NEW(t \mid f).\,]\!] \boxed{\text{TD}}\Rightarrow E_3$

where

$E_3 = \mathit{Add\text{-}Type}(\,\mathbf{typedec}(bool,\ \mathbf{tags}[\,\mathbf{tag}(t,\{\}),\ \mathbf{tag}(f,\{\})]), E_2)$

i.e.

E_3 = ([**typedec**($bool$, **tags**[**tag**(t,{}), **tag**(f,{})])],
[],
[],
{},

$\{\}$,
$\{\}$,
$\{bool \mapsto$ **typeno**$(1), f \mapsto$ **consttag**$(1), t \mapsto$ **consttag**$(1)\}$,
$\{\}$,
$\{\}$
$\{\}$
)

Applying the transformations to the function NOR:

In order to apply rule IN we need the following rule applications

T1 : $E_3 \vdash [\![bool]\!] \overset{}{=}\!\boxed{\mathrm{T}}\!\Rightarrow$ *Find-Type*$(bool, E_3) =$ **typeno**(1)

and
: Add-Signal(**signaldec**(in1, **typeno**(1), **input**), **joined**, E_3) = E_4

where

E_4 = ([**typedec**(bool, **tags**[**tag**(t,$\{\}$), **tag**(f,$\{\}$)])],
[],
[**signaldec**(in1, **typeno**(1), **input**)],
$\{\}$,
$\{\}$,
$\{\}$,
$\{bool \mapsto$ **typeno**$(1), f \mapsto$ **consttag**$(1), t \mapsto$ **consttag**$(1)\}$,
$\{\}$,
$\{in1 \mapsto$ **sig**$(1,$ **typeno**$(1),$ **joined**$)\}$
$\{\}$
)

also
T1 : $E_4 \vdash [\![bool]\!] \overset{}{=}\!\boxed{\mathrm{T}}\!\Rightarrow$ *Find-Type*$(bool, E_3) =$ **typeno**(1)

and
: Add-Signal(**signaldec**(in2, **typeno**(1), **input**), **joined**, E_4) = E_5

where

E_5 = ([**typedec**(bool, **tags**[**tag**(t,$\{\}$), **tag**(f,$\{\}$)])],
[],
[**signaldec**(in2, **typeno**(1), **input**), **signaldec**(in1, **typeno**(1), **input**)],
$\{\}$,
$\{\}$,
$\{\}$,
$\{$bool $\mapsto$ **typeno**(1), f $\mapsto$ **consttag**(1), t $\mapsto$ **consttag**(1)$\}$,
$\{\}$,
$\{$in2 $\mapsto$ **sig**(2, **typeno**(1), **joined**), in1 $\mapsto$ **sig**(1, **typeno**(1), **joined**)$\}$
$\{\}$

)

the premises of IN are now satisfied thus

IN : $E_3 \vdash [\![input]\!] \boxed{\text{IN}}\!\Rightarrow$ **types**[**typeno**(1), **typeno**(1)], E_5

and in order to satisfy the second premise of FD1 we need

T1 : $E_5 \vdash [\![bool]\!] \boxed{\text{T}}\!\Rightarrow$ **typeno**(1)

The evaluation of the CASE clause can now proceed :-

input signals

U1 : $E_5 \vdash [\![in1]\!] \boxed{\text{U}}\!\Rightarrow$ **signal**(1): **typeno**(1), E_5

U1 : $E_5 \vdash [\![in2]\!] \boxed{\text{U}}\!\Rightarrow$ **signal**(2): **typeno**(1), E_5

thus

CC2 : $E_5 \vdash [\![(in1,in2)]\!] \boxed{\text{CC}}\!\Rightarrow$ **units**[**signal**(1), **signal**(2)]: **types**[**typeno**(1), **typeno**(1)], E_5

hence

U11 : $E_5 \vdash [\![(in1,in2)]\!] \boxed{\text{U}}\!\Rightarrow$ **units**[**signal**(1), **signal**(2)]: **types**[**typeno**(1), **typeno**(1)], E_5

Evaluation of case arm alternatives:

CS4 : $E_5 \vdash [\![f]\!] \boxed{\text{CS}}\!\Rightarrow$ **enum**$(1,2)$, **typeno**(1)

CS4 : $E_5 \vdash [\![t]\!] \boxed{\text{CS}}\!\Rightarrow$ **enum**$(1,1)$, **typeno**(1)

combining these last two results gives

CS6 : $E_5 \vdash [\![(f,t)]\!] \boxed{\text{CS}}\!\Rightarrow$ **constsets**[**enum**$(1,2)$, **enum**$(1,1)$]: **types**[**typeno**(1), **typeno**(1)]

and together with

U2 : $E_5 \vdash [\![f]\!] \boxed{\text{U}}\!\Rightarrow$ **enum**$(1,2)$: **typeno**(1), E_5

gives

CA : $E_5 \vdash [\![(f,t){:}f)]\!] \boxed{\text{CA}}\!\Rightarrow$ **constsets**[**enum**$(1,2)$, **enum**$(1,1)$], **enum**$(1,2)$:
types[**typeno**(1), **typeno**(1)], **typeno**(1), E_5

Similarly

CA : $E_5 \vdash [\![(t,f){:}f)]\!] \boxed{\text{CA}}\!\Rightarrow$ **constsets**[**enum**$(1,1)$, **enum**$(1,2)$], **enum**$(1,2)$:
types[**typeno**(1), **typeno**(1)], **typeno**(1), E_5

CA : $E_5 \vdash [\![(t,t){:}f)]\!] \boxed{\text{CA}}\!\Rightarrow$ **constsets**[**enum**$(1,1)$, **enum**$(1,1)$], **enum**$(1,2)$:
types[**typeno**(1), **typeno**(1)], **typeno**(1), E_5

the ELSE clause of the CASE statement gives

U2 : $E_5 \vdash [\![t]\!] \boxed{\text{U}}\!\Rightarrow$ **enum**$(1,1)$: **typeno**(1), E_5

thus the complete CASE statement becomes

CC1 : $E_5 \vdash [\![CASE....ESAC]\!] \boxed{\text{CC}}\!\Rightarrow$
caseclause(**units**[**signal**(1), **signal**(2)],
[**case**(**constsets**[**enum**$(1,2)$, **enum**$(1,1)$], **enum**$(1,2)$),
case(**constsets**[**enum**$(1,1)$, **enum**$(1,2)$], **enum**$(1,2)$),

case(**constsets**[**enum**$(1,1)$, **enum**$(1,1)$], **enum**$(1,2)$)],
enum$(1,1)$, **typeno**(1)): **typeno**(1), E_5

Bringing the above results together for the complete function gives:

FD1 : $E_3 \vdash [\![FN\ NOR = \ldots.]\!] \boxed{\text{FD}}\Rightarrow E_6$

where

E_6 = Add-Fn (**fndec** (NOR, **types**[**typeno**(1), **typeno**(1)],
[**signaldec**(in2, **typeno**(1), **input**), **signaldec**(in1, **typeno**(1), **input**)],
typeno(1),
caseclause(**units**[**signal**(1), **signal**(2)],
[**case**(**constsets**[**enum**(1,2), **enum**(1,1)], **enum**(1,2)),
case(**constsets**[**enum**(1,1), **enum**(1,2)], **enum**(1,2)),
case(**constsets**[**enum**(1,1), **enum**(1,1)], **enum**(1,2))],
enum(1,1), **typeno**(1)),
E_3)

i.e

E_6 = ([**typedec**(bool, **tags**[**tag**(t,{}), **tag**(f,{})])],
[**fndec**(NOR,
types[**typeno**(1), **typeno**(1)],
[**signaldec**(in2, **typeno**(1), **input**), **signaldec**(in1, **typeno**(1), **input**)],
typeno(1),
caseclause(**units**[**signal**(1), **signal**(2)],
[**case**(**constsets**[**enum**(1,2), **enum**(1,1)], **enum**(1,2)),
case(**constsets**[**enum**(1,1), **enum**(1,2)], **enum**(1,2)),
case(**constsets**[**enum**(1,1), **enum**(1,1)], **enum**(1,2))],
enum(1,1),
typeno(1))
]
[],
{},
{NOR $\mapsto$ 1},
{},
{bool $\mapsto$ **typeno**(1), f $\mapsto$ **consttag**(1), t $\mapsto$ **consttag**(1)},
{},
{}
{}
)

This completes the environment for the function NOR. Now applying the transformations to DEL gives

T1 : $E_6 \vdash [\![bool]\!] \boxed{\text{T}}\Rightarrow$ *Find-Type*$(bool, E_6) =$ **typeno**(1)

and

: Add-Signal(**signaldec**(in, **typeno**(1), **input**), **joined**, E_6) = E_7

where

$$
\begin{aligned}
E_7 \;=\; & E_6 \dagger (\, [],\\
& [],\\
& [\,\mathbf{signaldec}(\text{in},\ \mathbf{typeno}(1),\ \mathbf{input})],\\
& \{\},\\
& \{\},\\
& \{\},\\
& \{\},\\
& \{\},\\
& \{\text{in} \mapsto\ \mathbf{sig}(1,\ \mathbf{typeno}(1),\ \mathbf{joined})\}\\
& \{\}\\
&)
\end{aligned}
$$

hence

IN : $E_6 \vdash [\![input]\!] \dashv\!\boxed{\text{IN}}\!\Rightarrow\ \mathbf{typeno}(1),\ E_7$

T1 : $E_7 \vdash [\![bool]\!] \dashv\!\boxed{\text{T}}\!\Rightarrow\ \mathbf{typeno}(1)$

Evaluation of the function body:

C3 : $E_7 \vdash [\![t]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow\ \mathbf{enum}(1,1):\ \mathbf{typeno}(1)$

BI3 : $E_7 \vdash [\![initial]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow\ \mathbf{enum}(1,1):\ \mathbf{typeno}(1)$

BI3 : $E_7 \vdash [\![ambig]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow\ \mathbf{enum}(1,1):\ \mathbf{typeno}(1)$

BI3 : $E_7 \vdash [\![DELAY(t,1,t,1)]\!]\,\{\,\mathbf{typeno}(1),\mathbf{typeno}(1)\} \dashv\!\boxed{\text{BI}}\!\Rightarrow\ \mathbf{delay}(\,\mathbf{enum}(1,1),\,1,\ \mathbf{enum}(1,1),\,1)$

Combining the above gives:

FD2 : $E_6 \vdash [\![FN\ DEL = \ldots.\ ]\!] \dashv\!\boxed{\text{FD}}\!\Rightarrow E_8$

where

E_8 = Add-Fn(**fndec**(DEL, **typeno**(1), [ ], **typeno**(1),
delay((**enum**(1,1), 1, **enum**(1,1), 1), E_6)

i.e.

$$
\begin{aligned}
E_8 \;=\; & E_6 \dagger (\, [],\\
& [\,\mathbf{fndec}(\text{DEL},\ \mathbf{typeno}(1),\\
& \qquad [],\ \mathbf{typeno}(1),\ \mathbf{delay}((\,\mathbf{enum}(1,1),\,1,\ \mathbf{enum}(1,1),\,1)\\
&]\\
& [],\\
& \{\},\\
& \{\text{DEL} \mapsto 2\},\\
& \{\},\\
& \{\},\\
& \{\},
\end{aligned}
$$

{}
{}
)

Finally given that

$E \vdash [\![TYPE\ bool = NEW(t \mid f).\ ]\!] \boxed{\text{TD}}\Rightarrow E_3$

$E_3 \vdash [\![FN\ NOR = ...]\!] \boxed{\text{FD}}\Rightarrow E_6$

$E_6 \vdash [\![FN\ DEL = ...\ ]\!] \boxed{\text{FD}}\Rightarrow E_8$

we have

CL : $E \vdash [\![Closure]\!] \boxed{\text{CL}}\Rightarrow E_8$

and so

KERNEL : KERNEL ($[\![closure]\!]$) = (E_8.Typedecs, E_8.Fndecs)

where

E_8.Typedecs = [**typedec**(bool, **tags**[**tag**(t,{}), **tag**(f,{})])]

and

```
E8.Fndecs   = [ fndec(NOR,
                      types[ typeno(1), typeno(1)],
                      [ signaldec(in2, typeno(1), input), signaldec(in1, typeno(1), input)],
                      typeno(1),
                      caseclause( units[ signal(1), signal(2)],
                         [ case( constsets[ enum(1,2), enum(1,1)], enum(1,2)),
                           case( constsets[ enum(1,1), enum(1,2)], enum(1,2)),
                           case( constsets[ enum(1,1), enum(1,1)], enum(1,2))],
                           enum(1,1),
                           typeno(1) ),
                fndec(DEL,
                      typeno(1),
                      [],
                      typeno(1),
                      delay(( enum(1,1), 1, enum(1,1), 1)
                ]
```

4.8 A Note on Dynamic Semantics

In this chapter we have presented the static semantics of the **Kernel**. In subsequent chapters the dynamic semantics for a simply typed subset of the **Kernel**, i.e. BKE, will be presented. The dynamic semantics for the complete **Kernel** still needs completing. In particular the handling of query-values, i.e. ?*type*, needs to be addressed. Such query-values could be thought of as denoting indeterminate signal values.

Thus a dynamically evaluated signal which has the value ?*type* could be taken to represent a signal with two or more non-deterministic values. Then the query-value can be thought of as an approximation to these values in order to allow a deterministic simulation.

Chapter 5

Boolean Kernel ELLA

This chapter introduces the current notation, Boolean Kernel ELLA (or *BKE*) for which semantics have been provided and issues of verification has been considered.

5.1 Syntax of *BKE*

We give a composite syntax for a simple ELLA-like language called *BKE*. It extends the LD language (i.e. Logic with Delay) from [BGM91] by including a "let" expression to allow local (recursive) bindings of variables and function declarations to provide encapsulation and modularity of designs.

BKE is related to Core ELLA [HM91] through its inclusion of function declarations and delays. It differs from Core ELLA in that "let" bindings can be recursive and that it does not include the ELLA model of data types. It can be thought of as a syntactic representation for an almost boolean version of the kernel described in the preceding chapter.

As usual, the convention for a composite syntax are that non-terminals correspond to types and indicated by names having their first letter in capitals. Instances of these types which appear on the right-hand side of productions have the same name, but are written in lower-case.

So, the composite syntax for *BKE* is as follows:

$$
\begin{array}{rcl}
\mathit{Closure} & ::= & \mathit{fndecl}_1, \cdots, \mathit{fndecl}_{k+1}. \\
\mathit{Fndecl} & ::= & \texttt{fn}\ \mathit{fnname}(\mathit{var}_1, \cdots, \mathit{var}_n) = \mathit{expr} \\
\mathit{Binding} & ::= & \mathit{bind}_1, \cdots, \mathit{bind}_{k+1} \\
\mathit{Bind} & ::= & \mathit{var}\ =\ \mathit{expr} \mid (\mathit{var}_1, \cdots, \mathit{var}_n) =\ \mathit{expr} \\
\mathit{Expr} & ::= & \mathit{const} \mid \mathit{var} \mid \mathit{let} \mid \mathit{row} \mid \mathit{case} \mid \mathit{apply} \mid \mathit{delays} \mid \mathit{sample} \mid \mathit{biop} \\
\mathit{Const} & ::= & 0 \mid 1 \\
\mathit{Var} & ::= & \mathit{id} \\
\mathit{Let} & ::= & \texttt{let}\ \mathit{binding}\ \texttt{in}\ \mathit{expr} \\
\mathit{Row} & ::= & (\mathit{expr}_1,\ \cdots, \mathit{expr}_{k+1}) \\
\mathit{Case} & ::= & \texttt{case}\ \mathit{row}\ \texttt{of}\ \mathit{caseitem}_1,\ \cdots,\ \mathit{caseitem}_{k+1}\ \texttt{else}\ \mathit{expr}\ \texttt{end} \\
\mathit{CaseItem} & ::= & \mathit{chooser} \rightarrow\ \mathit{expr} \\
\mathit{Chooser} & ::= & (\mathit{choice}_1,\ \cdots, \mathit{choice}_{k+1}) \\
\mathit{Choice} & ::= & \texttt{X}\ \mid \mathit{const} \\
\mathit{Apply} & ::= & \mathit{fnname}(\mathit{expr}_1,\ \cdots,\ \mathit{expr}_n) \\
\mathit{Delays} & ::= & \mathit{idelay}(\mathit{expr}, \mathit{const}, \mathit{nat}) \mid \mathit{delay}(\mathit{expr}, \mathit{const}_1, \mathit{nat}_1, \mathit{const}_2, \mathit{nat}_2) \\
\mathit{Sample} & ::= & \mathit{sample}(\mathit{expr}, \mathit{const}, \mathit{nat}_1, \mathit{nat}_2) \\
\mathit{Infix} & ::= & \mathit{expr}_1\ \mathit{binop}\ \mathit{expr}_2 \\
\mathit{Binop} & ::= & \mathit{not} \mid \mathit{or} \\
\mathit{Not} & ::= & \neg\, \mathit{expr} \\
\mathit{Or} & ::= & \mathit{expr}_1\ +\ \mathit{expr}_2 \\
\mathit{FnName} & ::= & \mathit{id} \\
\mathit{Nat} & ::= & \mathbb{N}
\end{array}
$$

In the production for Choice, the `X` symbol stands for "don't care".

5.1.1 Intuitive semantics of *BKE*

A *BKE* description, or closure, consists of a list of function declarations. Implicit in the model underlying *BKE* is a global clock, on the ticks of which the values of input signals are examined. Thus the model is synchronous and based upon states, rather than events. Consider a call of one of the functions in the closure. At any given clock tick, the function body, together with the given inputs and function declarations, is used to determine the output behaviour. As in ELLA, the use of function definitions and instantiations provides the main technique used to modularise descriptions.

We illustrate this briefly by giving descriptions of an RS flip flop, shown in Figure 5.1, using both ELLA and *BKE*. The ELLA text for the RS flip flop is given:

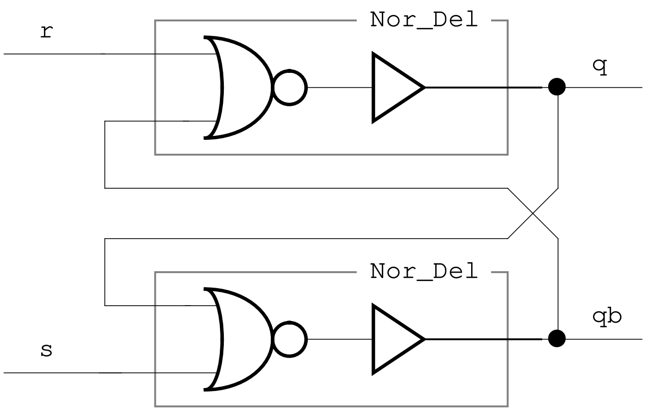

Figure 5.1: RS Flip Flop

```
TYPE bool = NEW(h | l).

FN BOOL_DELAY (bool) -> bool:
#initial value l, delay 1 unit: #
DELAY(l,1).

FN OR = (bool: x y) -> bool:
CASE (x,y) OF
  (l,l)           : l,
  (bool,h) | (h,bool): h
ESAC.

FN NOT = (bool: x) -> bool:
CASE x
OF l: h,
   h: l
ESAC.
```

```
FN NOR = (bool: x y) -> bool:
#defined in terms of NOT and OR: #
NOT(x OR y).

FN NOR_DEL = (bool: x y) -> bool:
BOOL_DELAY(NOR(x,y)).

FN RSFF = (bool: r s) -> (bool,bool):
BEGIN
  MAKE NOR_DEL: q qb.
  JOIN (r, qb) -> q,
       (q, s) -> qb.
  OUTPUT (q, qb)
END.
```

As an illustration of how ELLA is related to *BKE*, an intuitively equivalent description in *BKE* would be:

fn NOR(x, y) = ¬(x + y),

```
fn BOOL_DELAY(y) = IDelay(y,0,1),

fn NOR_DELAY(x, y) = BOOL_DELAY(NOR(x,y)),

fn RSFF(r s) =
  let q  = NOR_DEL(r,qb),
      qb = NOR_DEL(s,q)
  in (q,qb).
```

Thus the *BKE* description describes the behaviour of the devices output signal history, given the input signal history, and these can be represented in a very natural way as IO automata. This automaton semantics is illustrated in section 6.2.2.1

5.1.2 ELLA Features not covered by *BKE*

In this preliminary version of *BKE* it has not yet been possible to include the RAM primitive operation, the full typing mechanism and the unspecified value '?' available in Core ELLA. For all but the last feature, it is clear how to extend the present *BKE* definitions and to define special function objects to include these features (see Chapter 6).

The exact interpretation of '?' is more difficult to ascertain. In [BGHT90], '?' was interpreted as an extra value included in each data type. In §4.8 above it is interpreted as a signal which may be any one of the set of possible values of a particular type, allowing non-determinism to be handled in ELLA. In [Com90a], the following roles for '?' are identified:

a) As an input value to a CASE expression, it acts as an unknown but legal value of the appropriate type. It will match any appropriate type name in a chooser-value during choice determination. For example, an input of '?' would match the chooser-value 'bool', but would produce a '?' when tested against the chooser-value '(0 | 1)',

b) Output from an under-determined CASE expression (i.e. if an input does not match any chooser-value, and there is no ELSE clause),

c) A 'don't care' input value for simulation,

d) A 'don't care' value for delay, and sequence variable initial values,

e) Output from ARITH (BIOP) expression when return integer value goes out of range.

If '?' were introduced into *BKE* then the definitions of the basic functions would be extended as follows:

$$\neg(?) \quad = ?$$

$$(1+?) = 1 \quad (0+?) \quad = ? \qquad (?+?) = ?$$

The '?' is acting here as an additional special value in the data type. Work has begun on investigating algebraic models of '?', one of which identifies the behaviour of '?' with a notion of *ambiguity*[Mon91].

Part III

Formal Semantics

Chapter 6

Automaton Semantics for Boolean Kernel ELLA

This chapter presents an automata-based compositional semantics for *BKE*. We first define the notion of IO automata as needed by the semantics and then develop the semantics in a compositional manner using these automata as semantic objects.

6.1 Introduction

Hardware designers need to know what their designs mean when expressed in existing hardware design languages. In particular, if a design has to satisfy certain logical properties given in a specification, a correspondence between the design language and the specification's properties must be clearly established. A mathematical semantics for the design language is therefore required to establish this.

Hardware systems are in principle finite state, i.e. having bounded memory. Automated verification methods have been developed for finite state programs [GB88, BFG89b], so that the properties of state machines, such as equivalence, can be determined automatically. Hence, by semantically mapping the specification and implementation languages for hardware into state machines, such automatic verification techniques can be applied to hardware systems. This indicates that existing hardware design languages need semantics from which a state machine description can be easily extracted for each design. In the following two chapters we present two different semantic presentations that enjoy this property. In this chapter a semantics for Boolean Kernel ELLA is based directly on state machine descriptions, i.e. the semantic objects *are* IO Automata — a particular form of transition-labelled finite state machine. Thus, the semantic objects themselves can be directly used for verification purposes.

The next chapter, however, presents an alternative semantics, constructed by using an inference rule notation such as SOS[Plo81, Hen90]. This technique has been successfully applied to describe concurrent systems[Mil90, Hen88], general-purpose programming languages[HMT90]. It also forms the basis for a verification system for finite state programs[GB88, BFG89b]. The semantics described were, in fact, our initial steps at formalising particular aspects of signal computation performed at the hardware level. Two simplified languages are first introduced with the purpose of examining specific semantic features, such as timing delay, for example, before indicating steps required to obtain a semantics for *BKE*.

In the remaining sections of this chapter, we introduce IO automata and give a few motivational examples for their use in the semantics of hardware descriptions, then we present a compositional semantics for *BKE* using IO automata as the underlying semantic objects.

6.2 IO Automata and Hardware Components

The development of IO automata is part of the adaptation of well-known ideas from automata and language theory to the description of general system behaviour[VW86]. In some sense, they provide a generalisation of the *Mealy* finite state automaton from standard automata theory[HU79b, Shi87]. They have some similarity to the structures of the same name studied by Lynch and Tuttle [LT88].

More formally, we take a (basic) IO automaton as a 7-tuple comprising a set of states, an input alphabet, an output alphabet, an input/output-labelled transition relation, a distinguished set of states — the initial states, a distinguished set of states – the terminal states, and a set of subsets of states — the acceptance subsets. The latter component is a generalised form of Büchi-style acceptance condition and allows for the definition of restricted sets of infinite runs of the automata. For the purpose of giving meaning to ELLA components, this aspect can be ignored as we require all infinite runs to be accepting. In fact, in the definition below we omit both terminal and acceptance sets. However, acceptance subsets will reappear in chapter 12 where the automata are used with specification formalisms such as Propositional Linear Temporal Logic.

Using the meta language of VDM (see [Jon90] for an introduction to VDM) we have the following definitions.

$$
\begin{array}{lrcl}
\mathit{Trans} :: & \mathit{from} & : & \mathit{State} \\
 & \mathit{ivals} & : & \mathit{Name} \xrightarrow{m} \mathit{Val} \\
 & \mathit{ovals} & : & \mathit{Name} \xrightarrow{m} \mathit{Val} \\
 & \mathit{to} & : & \mathit{State}
\end{array}
$$

$$
\begin{array}{lrcl}
\mathit{IOA} :: & \mathit{states} & : & \mathit{State}\text{-}\mathbf{set} \\
 & \mathit{inputs} & : & \mathit{Name}\text{-}\mathbf{set} \\
 & \mathit{outputs} & : & \mathit{Name}\text{-}\mathbf{set} \\
 & \mathit{transitions} & : & \mathit{Trans}\text{-}\mathbf{set} \\
 & \mathit{start} & : & \mathit{State}\text{-}\mathbf{set}
\end{array}
$$

The invariant on *IOA* states that the start states are genuine states of our machine and that outputs may not also be inputs. Furthermore, the input and output mappings for each transition have the appropriate domain and the *from* and *to* states must be in the state set.

$$\mathit{inv\text{-}IOA} : \mathit{state}\text{-}\mathbf{set} \times \mathit{Name}\text{-}\mathbf{set} \times \mathit{Name}\text{-}\mathbf{set} \times \mathit{Trans}\text{-}\mathbf{set} \times \mathit{State}\text{-}\mathbf{set} \to \mathbb{B}$$

$$
\begin{array}{l}
\mathit{inv\text{-}IOA}(ss, i, o, ts, st) \quad \triangleq \quad st \subseteq ss \land i \cap o = \{\,\} \land \forall t \in ts \cdot \\
\quad \mathit{from}(i) \in ss \land \mathit{to}(i) \in ss \land \mathbf{dom}\, \mathit{ivals}(t) = i \land \mathbf{dom}\, \mathit{ovals}(t) = o
\end{array}
$$

6.2.1 IO Automaton Examples

6.2.1.1 NOR Gate

Consider a NOR gate (without delay) over the two valued set $\{0,1\}$. We model it by a single-state automaton with four transitions and is shown in Figure 6.1. For convenience, its transitions are repeated in tabular form in Table 6.1. The automaton has state set $\mathit{states} = \{n\}$, initial state set $\mathit{start} = \{n\}$ and value set $\mathit{Val} = \{0,1\}$. The labelling on the transitions in each automaton has been simplified for presentation purposes. Each mapping from label to value has been replaced by a value only, whose position determines the label to which it is assigned. In Figure 6.1 each label has the form ab/c, which represents input labels x, y and output label o, having values a, b and c respectively. From the given labellings, or transition table, it is clear that the output on any transition is the "nor" of the two inputs.

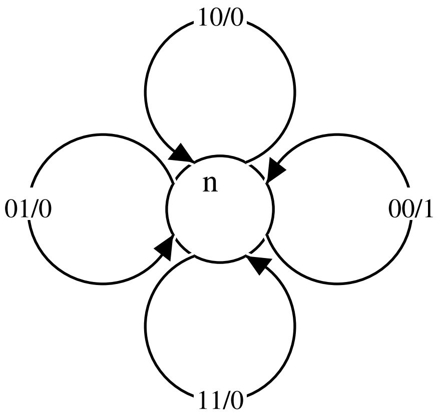

Figure 6.1: IO Automaton for Nor Gate

From	Input		Output	To
State	x	y	o	State
n	0	0	1	n
n	0	1	0	n
n	1	0	0	n
n	1	1	0	n

Table 6.1: Transition Table for Nor Gate

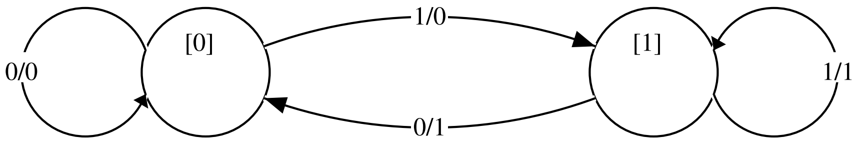

Figure 6.2: IO Automaton for Unit Delay

6.2.1.2 Unit Delay

Consider a unit delay function which delays the given input by one (notional) step. It can be modelled by a two-state automaton as shown in Figure 6.2. The initial state is state 0. Consider the possible transitions. Starting in the initial state, if the input is 0, then the output is 0 and the state does not change. If, however, the input had been 1, then 0 would have again been output but the new state would have been state 1. The idea is that state 0 corresponds to having input a 0 on the previous transition, i.e. all incident transitions have input 0, and its emergent transitions have output 0, thus effecting a unit delay on 0 inputs. Furthermore, the incident transitions of state 1 all have input 1 and the emergent transitions have output 1, thus effecting unit delay for 1 inputs.

6.2.2 Product IO Automata

Given two, or more, automata representing the behaviour of certain hardware components, we require some form of composition on automata which will correspond to "connecting" the hardware components together. For example, we might wish to connect a Unit Delay onto the output of a NOR gate forming a unit-delay-NOR gate. The composition required is a form of *product*.

There are, in fact, many ways of forming products of automata that may be of interest in automata-based verification. In the following semantic descriptions, we use a *causal* product as principal operation for constructing product automata. It is a "synchronous" product in that the resulting automaton produces a state transition only when each of the participants (in the product) have taken a transition of their own. In particular, it takes the usage of inputs and outputs between participants into account, so involving the notion of *causality* between participants. Furthermore, any input of one that happens to be an output of the other is hidden within the resulting product automaton. Therefore all communications exchanged between the participants are hidden from third-party external observers.

A VDM presentation of causal product follows.

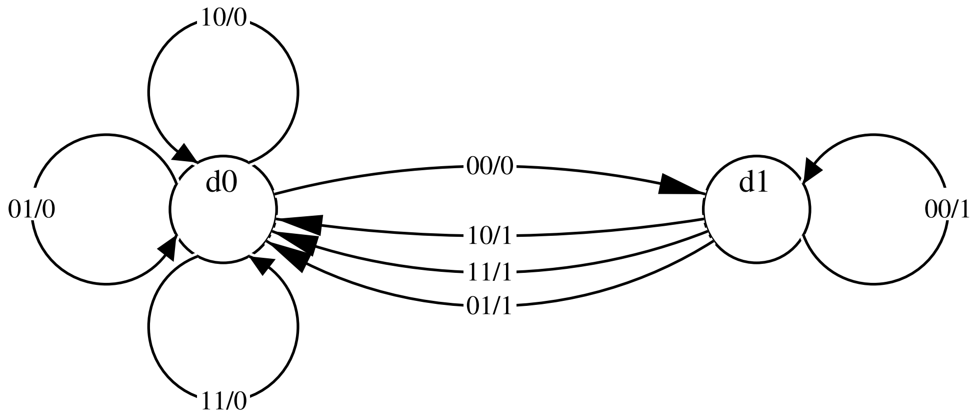

Figure 6.3: IO Automaton for Delay-Nor Gate

$$
\begin{array}{l}
\textit{causal-prod} : \textit{IOA} \times \textit{IOA} \rightarrow \textit{IOA} \\[6pt]
\textit{causal-prod}(a_1, a_2) \quad \underline{\triangle} \\
\qquad \textbf{let } \textit{mk-IOA}(ss_1, i_1, o_1, ts_1, st_1) = a_1 \textbf{ in} \\
\qquad \textbf{let } \textit{mk-IOA}(ss_2, i_2, o_2, ts_2, st_2) = a_2 \textbf{ in} \\
\qquad \textbf{let } o = o_1 \cup o_2 \textbf{ in} \\
\qquad \textbf{let } i = (i_1 \cup i_2) - o \textbf{ in} \\
\qquad \textbf{let } ts = \{\textit{mk-Trans}((s_1, s_2), im, om, (s_1', s_2')) \mid \\
\qquad\qquad\qquad \exists (s_1, im_1, om_1, s_1') \in ts_1, (s_2, im_2, om_2, s_2') \in ts_2 \cdot \\
\qquad\qquad\qquad \textbf{let } vals = im_1 \cup im_2 \cup om_1 \cup om_2 \textbf{ in} \\
\qquad\qquad\qquad im = i \lhd vals \wedge om = o \lhd vals\} \\
\qquad\quad \textbf{in} \\
\qquad \textit{mk-IOA}(ss_1 \times ss_2, i, o, \textit{new-ts}, st_1 \times st_2) \\[6pt]
\textbf{pre } \textit{outputs}(a_1) \cap \textit{outputs}(a_2) = \{\,\}
\end{array}
$$

6.2.2.1 Example: IO Automaton Construction for RS Flip Flop

To illustrate the automaton composition rule presented above, an IO automaton for an RS flip flop is composed from the automata representing two NOR gates, each having unit delay; the ELLA-text and *BKE* description for the RS flip flop appeared earlier in §5.1.1.

The undelayed NOR gate, of course, appeared above. To obtain a unit-delay-NOR gate automaton we contruct the causal product of a unit-delay automaton and the NOR gate automaton, forming a two-state machine. The unit-delay automaton appeared above in Figure 6.2. In §6.3, we will present general schema for various forms of delay functions. The IO automaton for the unit-delay-NOR gate and its equivalent transition table are are shown in Figure 6.3 and Table 6.2. The automaton has state set $states = \{d0, d1\}$, initial state set $start = \{d0\}$ and value set $Val = \{0, 1\}$. Output from NOR gate automaton is connected to the input of the delay automaton.

From State	Input x	Input y	Output o	To State
d0	0	0	0	d1
d0	0	1	0	d0
d0	1	0	0	d0
d0	1	1	0	d0
d1	0	0	1	d1
d1	0	1	1	d0
d1	1	0	1	d0
d1	1	1	1	d0

Table 6.2: Transition Table for Delay-Nor Gate

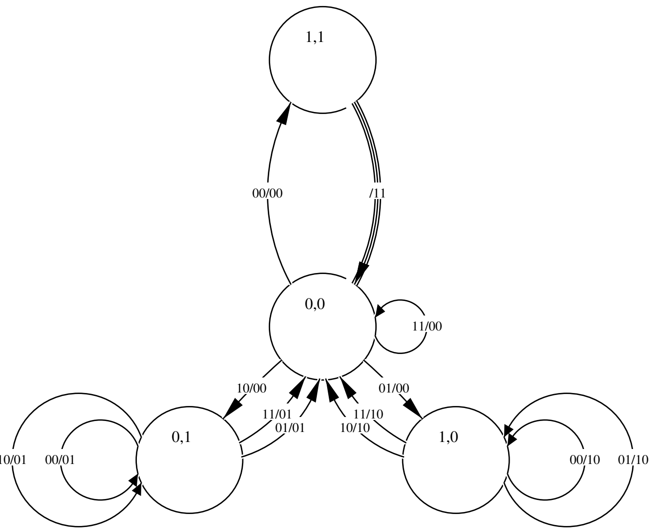

Figure 6.4: IO Automaton for RS Flip Flop

	x	y	o			x	y	o		Conflict		
From	x1	y1	o1	To	From	x2	y2	o2	To	Removal		
	r	O2	O1			O1	s	O2		A	B	C
D0	0	0	0	D1	d0	0	0	0	d1			Z
D0	0	1	0	D0	d0	0	0	0	d1	X		
D0	1	0	0	D0	d0	0	0	0	d1			Z
D0	1	1	0	D0	d0	0	0	0	d1	X		
D1	0	0	1	D1	d0	0	0	0	d1		Y	
D1	0	1	1	D0	d0	0	0	0	d1	X		
D1	1	0	1	D0	d0	0	0	0	d1		Y	
D1	1	1	1	D0	d0	0	0	0	d1	X		
D0	0	0	0	D1	d0	0	1	0	d0			Z
D0	0	1	0	D0	d0	0	1	0	d0	X		
.	.	.	.	.	.	.	.	.	.			
.	.	.	.	.	.	.	.	.	.			
.	.	.	.	.	.	.	.	.	.			
D0	0	0	0	D1	d1	1	1	1	d0	X		
D0	0	1	0	D0	d1	1	1	1	d0		Y	
D0	1	0	0	D0	d1	1	1	1	d0	X		
D0	1	1	0	D0	d1	1	1	1	d0		Y	
D1	0	0	1	D1	d1	1	1	1	d0	X		
D1	0	1	1	D0	d1	1	1	1	d0			Z
D1	1	0	1	D0	d1	1	1	1	d0	X		
D1	1	1	1	D0	d1	1	1	1	d0			Z

Table 6.3: Calculation of Transition Relation of RS Flip Flop Automaton

Table 6.3 shows the first stage in calculating the transition table for the RS flip flop automaton, by giving all combinations of transitions from the two delay-NOR gate automata. The first automaton instance has state labels $D0$ and $D1$, input labels $x1$ and $y1$, and output label $o1$; the second instance has state labels $d0$ and $d1$, input labels $x2$ and $y2$ and output label $o2$.[1]

Finally, two instances of a delay-NOR gate automaton are composed, using the causal product, to form the RS flip flop machine, shown in Figure 6.4.

The two inputs $x2$ and $y1$ from the component automata are hidden and renamed $O1$ and $O2$ respectively when the causal product is formed. Conflicting transitions are then removed from the resulting transition table: those transitions marked X in column A of Table 6.3 are deleted because of conflicts between input $y1$ and output $O2$, those marked Y in column B are deleted because of conflicts between input $x2$ and $O1$, leaving the remaining valid transitions marked Z in column C, shown in Table 6.4.

The now redundant information for the renamed inputs $x2$ and $y1$ is removed, and separate state labels Di and dj are paired to form the resulting single state label (i,j). The transitions which remain in the automaton are shown in Figure 6.4. A transition ab/cd in the diagram represents inputs r and s, outputs $O1$ and $O2$, having the values a, b, c, and d respectively. The triple arrow which emerges from State $(1,1)$ is labelled with $/11$, which means that for any input received in State $(1,1)$, then the output $O1 = 1, O2 = 1$ will be produced when going to State $(0,0)$.

From	r	(O2)	O1	To	From	(O1)	s	O2	To	A	B	C
D0	0	0	0	D1	d0	0	0	0	d1			Z
D0	0	0	0	D1	d0	0	1	0	d0			Z
D0	1	0	0	D0	d0	0	0	0	d1			Z
D0	1	0	0	D0	d0	0	1	0	d0			Z
D0	0	1	0	D0	d1	0	0	1	d1			Z
D0	0	1	0	D0	d1	0	1	1	d0			Z
D0	1	1	0	D0	d1	0	0	1	d1			Z
D0	1	1	0	D0	d1	0	1	1	d0			Z
D1	0	0	1	D1	d0	1	0	0	d0			Z
D1	0	0	1	D1	d0	1	1	0	d0			Z
D1	1	0	1	D0	d0	1	0	0	d0			Z
D1	1	0	1	D0	d0	1	1	0	d0			Z
D1	0	1	1	D0	d1	1	0	1	d0			Z
D1	0	1	1	D0	d1	1	1	1	d0			Z
D1	1	1	1	D0	d1	1	0	1	d0			Z
D1	1	1	1	D0	d1	1	1	1	d0			Z

Table 6.4: Modified Transition Table for RS Flip Flop

[1] In an implementation of the causal product operation, the complete transition tables are not constructed in this way. This is done here purely for explanatory purposes.

6.3 Semantics of *BKE*

We now present a direct semantics, based on IO automata, for *BKE*. A related style of semantics was presented by Milner for a simple parallel programming language in [Mil90]. The notation used follows the style of C. B. Jones' presentation of a denotational semantics for a Pascal-like programming language in [JS90]. We will present the semantics of *BKE* along with the abstract syntax, in a "bottom-up" fashion with the simplest constructs coming first. We ignore the static semantics of *BKE* in our discussion, so we simply assume that functions are defined before use and are called with the proper number of arguments, and so on.

6.3.1 Semantic Objects

BKE, like ELLA, is an expression-oriented language. Thus, the semantic objects must provide a uniform mechanism for naming the results of expressions. Furthermore, this naming mechanism must allow for the multiple-valued Boolean results of *BKE*, e.g. collateral of Boolean valued expressions. This is achieved by using a set of distinct internal names for outputs. For simplicity, we take this set as the positive integers. So, the IO automaton introduced and exemplified in the preceding sections is restricted as follows to give the basic semantic object *SemObj*.

$$Name = Id \cup \mathbb{Z} \setminus \{0\}$$

$$SemObj = IOA$$

where

$$inv\text{-}SemObj : IOA \to \mathbb{B}$$

$$inv\text{-}SemObj(mk\text{-}IOA(ss, i, o, ts, st)) \quad \triangleq \quad \exists n \in \mathbb{N} \cdot o = \{1, \ldots, n\}$$

6.3.1.1 Function objects

Function application will be achieved by taking products of automata whose input and/or output ports have been suitably renamed to effect connections. A "compiled" function, or *function object*, is an automaton, some of whose inputs are to be connected to the outputs of the arguments in a call of the function (see section 6.3.2.6).

We will represent a function object of arity n by a *SemObj* whose input set will include the negative integers -1, …, -n acting as formal argument names. To call the function, we attach the outputs of the actual arguments to the numbered inputs. Note that negative numbers are not legal program names. In this way, we can bind the actual arguments in a function application to the formal parameters without any danger of a name clash with the inputs.

6.3.2 Expressions

The first syntactic category we consider is the *Expression*.

$$Expression = Const \cup Var \cup Let \cup Row \cup Case \cup Funapp \cup Delays \cup Sample \cup Not \cup Infix$$

The meaning of an expression will be given relative to an *environment*, which captures the function definitions in effect at the point where the expression occurs.

$$Env = Name \xrightarrow{m} SemObj$$

The denotation of an expression in a given environment will be a *SemObj*. The semantic function for an expression simply dispatches according to the expression type as follows:-

$$MExpression : Expression \to Env \to SemObj$$

$$
\begin{array}{lll}
MExpression[\![e]\!]\sigma & \triangleq & \textbf{cases } e \textbf{ of} \\
 & & \begin{array}{ll}
 \textit{mk-Const}(v) & \to MConst[\![e]\!] \\
 \textit{mk-Var}(v) & \to MVar[\![e]\!] \\
 \textit{mk-Let}(bd, e_1) & \to MLet[\![e]\!]\sigma \\
 \textit{mk-Row}(els) & \to MRow[\![els]\!]\sigma \\
 \textit{mk-Case}(rw, itms, else) & \to MCase[\![e]\!]\sigma \\
 \textit{mk-Funapp}(f, args) & \to MFunapp[\![e]\!]\sigma \\
 \textit{mk-IDelay}(e_1, v, n) & \to MIDelay[\![e]\!]\sigma \\
 \textit{mk-Delay}(e_1, v_1, n_1, v_2, n_2) & \to MDelay[\![e]\!]\sigma \\
 \textit{mk-Sample}(e_1, v, n_1, n_2) & \to MSample[\![e]\!]\sigma \\
 \textit{mk-Not}(e_1) & \to MNot[\![e]\!]\sigma \\
 \textit{mk-Infix}(e_1, op, e_2) & \to MInfix[\![e]\!]\sigma
 \end{array} \\
 & & \textbf{end}
\end{array}
$$

6.3.2.1 Constants

A constant denotes a one-state automaton whose purpose is to produce the desired constant value whenever it is required. The automaton has a single state, no inputs, one output, and one transition which outputs the constant value.

$$Const :: Val$$

$$MConst : Const \to SemObj$$

$$MConst[\![\textit{mk-Const}(c)]\!] \triangleq \textit{const-machine}(c)$$

$$\textit{const-machine} : Val \to SemObj$$

$$\textit{const-machine}(c) \triangleq \textit{mk-IOA}(\{0\}, \{\,\}, \{1\}, \{\textit{mk-Trans}(0, \{\,\}, \{1 \mapsto c\}, 0)\}, \{0\})$$

6.3.2.2 Variables

A variable reference denotes a one-state machine whose transitions simply copy the (single) input value to the output. The sole purpose of such a machine is to record the variable's name in the input set.

$$Var :: Id$$

$$MVar : Var \to SemObj$$

$$MVar[\![\textit{mk-Var}(x)]\!] \triangleq \textit{var-machine}(x)$$

$$var\text{-}machine : Id \rightarrow SemObj$$

$$\begin{aligned}
&var\text{-}machine(x) \quad \triangleq \\
&\quad \textbf{let } ts = \{mk\text{-}Trans(0, \{x \mapsto v\}, \{1 \mapsto v\}, 0) \mid v \in Val\} \ \textbf{ in} \\
&\quad mk\text{-}IOA(\{0\}, \{x\}, \{1\}, ts, \{0\})
\end{aligned}$$

6.3.2.3 Recursive Let

The *Let* construct in *BKE* consists of a *Binding* and an *Expression*, which may refer to the bound variables of the *Binding*. To evaluate a *Let*, we evaluate the *Binding* and the *Expression* independently, form the causal product of the results, then restrict to the outputs of the given expression.

$$\begin{aligned}
Let \ :: \ \ & eqns \ : \ Binding \\
& return \ : \ Expression
\end{aligned}$$

$$MLet : Let \rightarrow Env \rightarrow SemObj$$

$$\begin{aligned}
&MLet[\![mk\text{-}Let(eqs, ret)]\!]\sigma \quad \triangleq \\
&\quad \textbf{let } expr\text{-}obj = MExpression[\![ret]\!]\sigma \ \textbf{ in} \\
&\quad \textbf{let } outs = outputs(expr\text{-}obj) \ \textbf{ in} \\
&\quad restrict(causal\text{-}product(Mbinding[\![eqns]\!]\sigma, expr\text{-}obj), outs)
\end{aligned}$$

Bindings A *Binding* in *BKE* consists of a non-empty sequence of equations either of the form $id = expr$, or $id_1, \ldots, id_n = expr$, where the identifiers on the left hand sides must all be distinct, though they may also appear in the expressions on the right hand sides. The meaning of a binding is a general *IOA* formed by evaluating each of the right hand sides, renaming the outputs to the left hand sides and taking the causal product of the resulting sequence of automata. We can think of this process as "solving" a set of simultaneous equations, since the only transitions permitted in the product are those whose input/output values satisfy the equations.

A *BindList* is a sequence of pairs which consist of a sequence of *Name* together with an *Expression* or an *SemObj*. The names must all be distinct.

$$\begin{aligned}
BindElem \ :: \ \ & lhs \ : \ Name^{+} \\
& rhs \ : \ Expression \cup SemObj
\end{aligned}$$

$$BindList = BindElem^{+}$$

$$inv\text{-}BindList : BindElem^{+} \rightarrow \mathbb{B}$$

$$inv\text{-}BindList(bl) \quad \triangleq \quad \forall i, j \cdot i \neq j \ \Rightarrow \ (\textbf{elems}\, lhs(bl(i)) \cap \textbf{elems}\, lhs(bl(j))) = \{\,\}$$

We represent a list of equations in *BKE* by a *Binding*. A *Binding* is a *BindList* whose left hand sides are all program identifiers and whose right hand sides are all expressions.

$$Binding \ :: \ BindList$$

$$inv\text{-}Binding : BindList \rightarrow \mathbb{B}$$

$$inv\text{-}Binding(mk\text{-}Binding(l)) \quad \triangleq \quad \forall x \in \textbf{elems}\, l \cdot rhs(x) \in Expression$$

$MBinding : Binding \rightarrow Env \rightarrow IOA$

$MBinding[\![mk\text{-}Binding(bdl)]\!]\sigma \quad \triangleq \quad bind(eval\text{-}rhs(bdl,\sigma))$

$eval\text{-}rhs : BindList \times Env \rightarrow BindList$

$$
\begin{array}{l}
eval\text{-}rhs(bdl,\sigma) \quad \triangleq \\
\quad \textbf{cases}\ bdl\ \ \textbf{of} \\
\quad [mk\text{-}BindElem(x,e)] \qquad \rightarrow [mk\text{-}BindElem(x,MExpression[\![e]\!]\sigma)] \\
\quad [mk\text{-}BindElem(x,e)] \frown l \rightarrow mk\text{-}BindElem(x,MExpression[\![e]\!]\sigma) \frown eval\text{-}rhs(l) \\
\quad \textbf{end}
\end{array}
$$

$\textbf{pre}\ \forall x \in \textbf{elems}\, bdl \cdot rhs(x) \in Expression$

Since the binding operation is also used in section 6.3.2.6, we need to distinguish between *Bindings* and *evaluated* binding lists, in which the expressions have already been replaced by automata.

$EvBindList = BindList$

where

$inv\text{-}EvBindList : BindElem^{+} \rightarrow \mathbb{B}$

$inv\text{-}EvBindList(l) \quad \triangleq \quad \forall x \in \textbf{elems}\, l \cdot rhs(x) \in SemObj$

$bind : EvBindList \rightarrow IOA$

$$
\begin{array}{ll}
bind(bdlist) \quad \triangleq & \textbf{cases}\ bdlist\ \ \textbf{of} \\
& [mk\text{-}BindElem(x,a)] \qquad \rightarrow rename(a,x) \\
& l \frown [mk\text{-}BindElem(x,a)] \rightarrow causal\text{-}prod(bind(l),rename(a,x)) \\
& \textbf{end}
\end{array}
$$

The function *rename* is similar to the *restrict* function of section 6.3.2.6. It is complicated, however, by the possibility that some of the new output names may already be input names of the automaton. In that case the offending output name must be removed from the set of inputs and only those transitions for which the output values agree with the relevant inputs are retained.

$rename : SemObj \times Name^{+} \rightarrow IOA$

$$
\begin{array}{l}
rename(a,xl) \quad \triangleq \\
\quad \textbf{let}\ mk\text{-}IOA(ss,i,o,ts,s) = a\ \ \textbf{in} \\
\quad \textbf{let}\ inds = \{j \mid xl(j) \in i\}\ \ \textbf{in} \\
\quad \textbf{let}\ new\text{-}ts = \{rename\text{-}trans(t,xl) \mid t \in ts \wedge \forall j \in inds \cdot ivals(t)(xl(j)) = ovals(t)(j)\}\ \ \textbf{in} \\
\quad mk\text{-}IOA(ss, i - \textbf{elems}\, xl, \textbf{elems}\, xl, new\text{-}ts, s)
\end{array}
$$

$\textbf{pre}\ outputs(a) = \textbf{inds}\, xl \wedge \textbf{card}\ (\textbf{elems}\, xl) = \textbf{len}\, xl$

$rename\text{-}trans : Trans \times Name^{+} \rightarrow IOA$

$$
\begin{array}{l}
rename\text{-}trans(t,xl) \quad \triangleq \\
\quad \textbf{let}\ mk\text{-}Trans(fr,iv,ov,to) = t\ \ \textbf{in} \\
\quad mk\text{-}Trans(fr, \textbf{elems}\, x \triangleleft iv, \{xl(i) \mapsto ov(i) \mid i \in \textbf{inds}\, xl\}, to)
\end{array}
$$

$\textbf{pre}\ \textbf{dom}\, ovals(t) = \textbf{inds}\, xl$

6.3.2.4 Rows

A row expression is denoted by a *SemObj* that has multiple outputs, e.g. $(e_1, \ldots, e_n)$ will be denoted by an automaton that produces n outputs, named 1 up to n. The semantic object is constructed by considering the row as a binding of the form $[(1, e_1), \ldots, (n, e_n)]$ and applying the meaning function for bindings. The function *number-outputs* creates the binding from the sequence of expressions representing the row.

$Row = Expression^+$

$MRow : Row \to Env \to SemObj$

$MRow[\![els]\!]\sigma \quad \triangleq \quad MBinding(number\text{-}outputs(els))$

$number\text{-}outputs : Expression^+ \to Binding$

$number\text{-}outputs(els) \quad \triangleq$
cases *els* **of**
$[\,] \qquad \to [\,]$
$el \frown [expr] \to [mk\text{-}BindElem([\mathbf{len}\, els], expr)] \frown number\text{-}outputs(el)$
end

6.3.2.5 Case expressions

Before giving the formal semantics of the *case* statement, we outline the intuition behind the definition. Suppose we have a *case* statement of the form

```
case (u1,...,uk) of
  (c11,...,c1k)  : e1
  (c21,...,c2k)  : e2
    ......
  (cn1,...,cnk)  : en
     else
              e
end
```

where the selector expression is a row of width k and there are n cases. Suppose also that each of the expressions ei evaluates to a row of width s. The selector expression is represented by an automaton *selector-ioa* with k outputs, of the form shown in figure 6.5. The number of inputs of *selector-ioa* will depend on the number of free variables in the expressions ui and is not relevant to this discussion.

Each of the eis is represented by an automaton with s outputs. The case mechanism is effected by connecting these automata to an automaton *chooser-ioa* which acts in the rôle of a multiplexor. This automaton, whose form is shown in figure 6.5, has $k + (n+1)s$ inputs, partitioned as follows. The first k are connected to the output of *selector-ioa*, to give the value of the selector expression. The next ns inputs are connected, in blocks of s, to the outputs from the automata corresponding to the ei's. The final block of s inputs is connected to the outputs of the automaton for e. The automaton *chooser-ioa* has just one state and acts as follows: if the first k inputs have values matching (ci1,...,cik) , then the inputs from the ith block of s inputs are copied to the output.

$ChoiceVal = Val \cup \{\mathrm{X}\}$

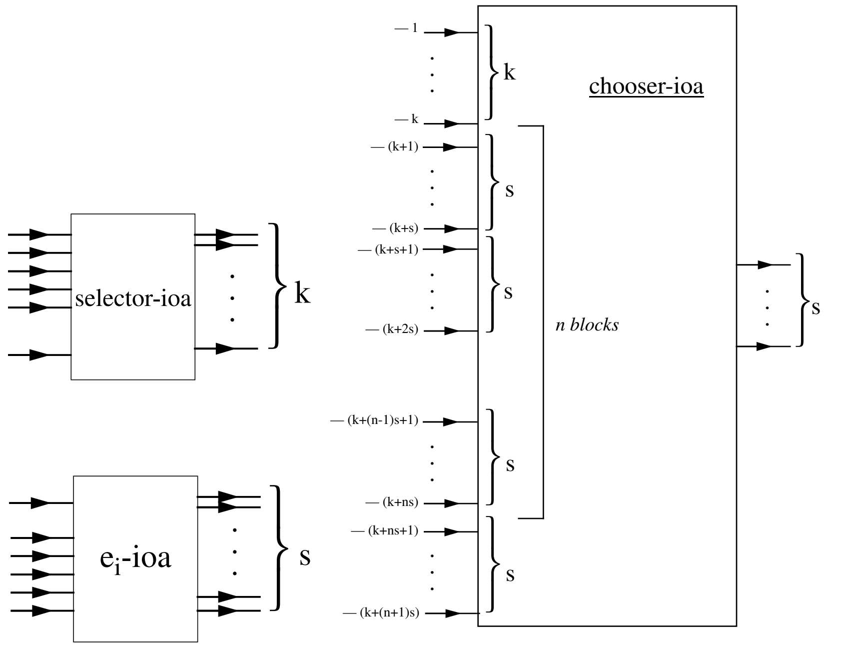

Figure 6.5: Case automaton components

$$\mathit{RowChoiceVal} = \mathit{ChoiceVal}^{+}$$

$$\begin{array}{lrcl}
\mathit{Case} :: & \mathit{selector} & : & \mathit{Row} \\
 & \mathit{items} & : & \mathit{CaseItem}^{+} \\
 & \mathit{else\text{-}row} & : & \mathit{Row}
\end{array}$$

$$\begin{array}{lrcl}
\mathit{CaseItem} :: & \mathit{chooser} & : & \mathit{RowChoiceVal} \\
 & \mathit{result} & : & \mathit{Row}
\end{array}$$

$$\mathit{MCase} : \mathit{Case} \rightarrow \mathit{Env} \rightarrow \mathit{SemObj}$$

$$\begin{array}{l}
\mathit{MCase}[\![\mathit{mk\text{-}Case}(\mathit{selector}, \mathit{itms}, \mathit{else\text{-}row})]\!]\sigma \quad \triangleq \\
\quad \textbf{let } k = \textbf{len}\, \mathit{selector} \textbf{ in} \\
\quad \textbf{let } n = \textbf{len}\, \mathit{itms} \textbf{ in} \\
\quad \textbf{let } s = \textbf{len}\, \mathit{result}(\textbf{hd}\, \mathit{itms}) \textbf{ in} \\
\quad \textbf{let } \mathit{selector\text{-}ioa} = \mathit{MRow}[\![\mathit{selector}]\!]\sigma \textbf{ in} \\
\quad \textbf{let } \mathit{chooser\text{-}ioa} = \mathit{MChList}(\mathit{get\text{-}choosers}(\mathit{itms})) \textbf{ in} \\
\quad \textbf{let } \mathit{choices\text{-}ioa} = \mathit{iter\text{-}causal\text{-}prod}(\mathit{get\text{-}choices\text{-}rename}(k, s, \mathit{itms}, \sigma)) \textbf{ in} \\
\quad \textbf{let } \mathit{else\text{-}ioa} = \mathit{rename}(\mathit{MExpression}[\![\mathit{else\text{-}row}]\!]\sigma, \mathit{outseq}(k, n, s)) \textbf{ in} \\
\quad \mathit{iter\text{-}causal\text{-}prod}([\mathit{selector\text{-}ioa}, \mathit{chooser\text{-}ioa}, \mathit{choices\text{-}ioa}, \mathit{else\text{-}ioa}])
\end{array}$$

MCase builds its result by taking the causal product of 4 automata. The first two of these are the automata *selector-ioa* and *chooser-ioa*, described above. The other two are *choices-ioa*, which is constructed by taking the causal product of the ei's with their outputs renamed to connect to the appropriate inputs of *chooser-ioa*, and *else-ioa*, the automaton corresponding to the action for the else clause, with its outputs suitably renamed.

$$MChList : RowChoiceVal^{+} \to IOA$$

$$\begin{array}{l}
MChList(rch\text{-}list) \quad \triangleq \\
\quad \textbf{let } k = \textbf{len}\, chooser(\textbf{hd}\, rch\text{-}list) \;\; \textbf{in} \\
\quad \textbf{let } n = \textbf{len}\, rch\text{-}list \;\; \textbf{in} \\
\quad \textbf{let } s = \textbf{len}\, result(\textbf{hd}\, rch\text{-}list) \;\; \textbf{in} \\
\quad \textbf{let } input\text{-}set = \{-(k+(n+1)s), \ldots, -1\} \;\; \textbf{in} \\
\quad \textbf{let } output\text{-}set = \{1, \ldots, s\} \;\; \textbf{in} \\
\quad \textbf{let } poss\text{-}alt\text{-}maps = \{-s, \ldots, -1\} \xrightarrow{m} Val \;\; \textbf{in} \\
\quad \textbf{let } poss\text{-}chooser\text{-}maps = \{-k, \ldots, -1\} \xrightarrow{m} Val \;\; \textbf{in} \\
\quad \textbf{let } choices\text{-}trans = \\
\qquad\qquad \bigcup\{build\text{-}transitions(mapify(rch\text{-}list[i], -1), poss\text{-}alt\text{-}maps, k+(i-1)s) \\
\qquad\qquad\quad \mid i \in \textbf{inds}\, rch\text{-}list\} \\
\quad \textbf{in} \\
\quad \textbf{let } choices\text{-}input\text{-}maps = \{ivals(tr) \mid tr \in choices\text{-}trans\} \;\; \textbf{in} \\
\quad \textbf{let } else\text{-}maps = \{m \in poss\text{-}chooser\text{-}maps \mid \neg\, \exists i \in choices\text{-}input\text{-}maps \cdot i \subseteq m\} \;\; \textbf{in} \\
\quad \textbf{let } else\text{-}trans = \\
\qquad\qquad \bigcup\{build\text{-}transitions(m, poss\text{-}alt\text{-}maps, k+n\times s) \mid m \in else\text{-}maps\} \;\; \textbf{in} \\
\quad mk\text{-}IOA(\{0\}, input\text{-}set, output\text{-}set, choices\text{-}trans \cup else\text{-}trans, \{0\})
\end{array}$$

The function *MChList*, used to build *chooser-ioa* is the core of the construction. The rôle of *chooser-ioa* is, on the basis of the values its first k inputs, to copy the appropriate block of s inputs to its outputs. The function *build-transitions* is used to build a set of transitions corresponding to a particular choice, with inputs constructed from *chooser-in-map* and a set of possible inputs, *in-maps*, from a block of s inputs, which are just copied across to the output.

It might appear that a large number of internal signals are generated in this construction; in fact they all disappear from the final automaton, whose inputs correspond to the free variables in the expression and output width is that of one of the choice actions.

$$build\text{-}transitions : (Name \xrightarrow{m} ChoiceVal) \times (\mathbb{Z} \xrightarrow{m} Val)\text{-}\textbf{set} \times \mathbb{N} \to Trans\text{-}\textbf{set}$$

$$\begin{array}{l}
build\text{-}transitions(chooser\text{-}in\text{-}map, in\text{-}maps, args\text{-}shift) \quad \triangleq \\
\quad \{mk\text{-}Trans(0, chooser\text{-}in\text{-}map \cup (m \circ shift(args\text{-}shift)), m \circ reflect, 0) \mid m \in in\text{-}maps\}
\end{array}$$

$$mapify : RowChoiceVal \times \mathbb{Z} \to (\mathbb{Z} \xrightarrow{m} Val)$$

$$mapify(rcv, porm) \quad \triangleq \quad \{porm \times i \mapsto rcv[i] \mid i \in \textbf{inds}\, rcv \wedge rcv[i] \neq X\}$$

$$shift : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$$

$$shift(sh)(n) \quad \triangleq \quad n - sh$$

$$reflect : \mathbb{Z} \to \mathbb{Z}$$

$$reflect(n) \quad \triangleq \quad -n$$

$iter\text{-}causal\text{-}prod : IOA^+ \to IOA$

$$
\begin{array}{lll}
iter\text{-}causal\text{-}prod(ioal) & \triangleq & \textbf{cases}\ ioal\ \textbf{of} \\
& & [ioa] \quad\quad\quad \to ioa \\
& & [ioa] \frown rest \to causal\text{-}product(ioa, iter\text{-}causal\text{-}prod(rest)) \\
& & \textbf{end}
\end{array}
$$

$get\text{-}choosers : CaseItem^* \to RowChoiceVal^*$

$get\text{-}choosers(ci\text{-}list) \quad \triangleq$

$$
\begin{array}{l}
\textbf{cases}\ ci\text{-}list\ \textbf{of} \\
[\,] \quad\quad\quad \to [\,] \\
[ci] \frown rest \to [chooser(ci)] \frown get\text{-}choosers(rest) \\
\textbf{end}
\end{array}
$$

$get\text{-}choices\text{-}rename \colon (init\text{-}shift \colon \mathbb{N}, out\text{-}len \colon \mathbb{N}, ci\text{-}list \colon CaseItem^+, \sigma \colon Env) \to ioa\text{-}list \colon IOA^+$

$\textbf{post}\ (\textbf{len}\ ioa\text{-}list = \textbf{len}\ ci\text{-}list) \wedge$

$\quad \forall i \in \textbf{inds}\ ci\text{-}list \cdot$

$\quad\quad ioa\text{-}list[i] = rename(MExpression[\![result(ci\text{-}list[i])]\!]\sigma, outseq(init\text{-}shift, i-1, out\text{-}len))$

$outseq \colon (first \colon \mathbb{N}1, blocks \colon \mathbb{N}, length \colon \mathbb{N}1) \to os \colon \mathbb{Z}^*$

$\textbf{post}\ \textbf{len}\ os = length \wedge$

$\quad \forall i \in \textbf{inds}\ os \cdot$

$\quad\quad os[i] = -(first + blocks \times length + i)$

An example automaton construction including a case statement is given in section 10.3.

6.3.2.6 Function application

To apply a function to a sequence of arguments, we look up the function definition in the environment. We will assume that the number of arguments in the call matches the number specified by the function object. We evaluate the arguments into a list of automata (using *eval-list*). We then apply the function object to the evaluated arguments by renaming their outputs to the names $-1, \ldots, -n$ and taking the product with the body of the function object (this is what the function *bind* does—see section 6.3.2.3). The function *number-autlist* takes a list $[a_1 \ldots a_n]$ and returns the list $[(-1, a_1) \ldots (-n, a_n)]$. Finally we restrict the output of our automaton to the outputs of the function.

$$
\begin{array}{llll}
Funapp & :: & func & : Name \\
& & args & : Expression^*
\end{array}
$$

$MFunapp : Funapp \to Env \to SemObj$

$MFunapp[\![mk\text{-}Funapp(f, args)]\!]\sigma \quad \triangleq \quad apply(\sigma(f), eval\text{-}list(args, \sigma))$

$eval\text{-}list : Expression^* \times Env \rightarrow Env$

$$
\begin{array}{lll}
eval\text{-}list(exps, \sigma) & \triangleq & \textbf{cases}\ exps\ \ \textbf{of} \\
& & [\,] \quad\quad\quad \rightarrow [\,] \\
& & [e] \frown rest \rightarrow [MExpression[\![e]\!]\sigma] \frown eval\text{-}list(rest, \sigma) \\
& & \textbf{end}
\end{array}
$$

$apply : SemObj \times SemObj^* \rightarrow SemObj$

$$
\begin{array}{l}
apply(fo, alist) \ \ \triangleq \\
\quad restrict(bind([([1, \ldots, \textbf{card}\, outputs(fo)], fo)] \frown number\text{-}autlist(alist)), outputs(fo))
\end{array}
$$

$number\text{-}autlist : SemObj^* \rightarrow (Name^+ \times SemObj)^*$

$$
\begin{array}{lll}
number\text{-}autlist(l) & \triangleq & \textbf{cases}\ l\ \ \textbf{of} \\
& & [\,] \quad\quad \rightarrow [\,] \\
& & l_1 \frown [a] \rightarrow number\text{-}autlist(l_1) \frown [([-\textbf{len}\, l], a)] \\
& & \textbf{end}
\end{array}
$$

$restrict : IOA \times Name\text{-}\textbf{set} \rightarrow SemObj$

$$
\begin{array}{l}
restrict(a, xs) \ \ \triangleq \\
\quad \textbf{let}\ new\text{-}ts = \{\mu(t, ovals \mapsto \{x \mapsto ovals(t)(x)\}) \mid t \in ts(a) \wedge x \in xs\}\ \ \textbf{in} \\
\quad \mu(a, transitions \mapsto new\text{-}ts)
\end{array}
$$

$\textbf{pre}\ xs \subseteq outputs(a)$

6.3.2.7 Delays

A delay operation in *BKE* applies a predefined IO automaton to the result of its expression operand. *BKE* contains two different delay types called ambiguity delay *Delay* and inertial delay *IDelay*[HWM90b]. This section will show how the IO automata are constructed for both; a different IO automaton is required for each delay duration and initial output value. There are also two special cases of ambiguity and inertial delay – transport delay $Delay_{trans}$ and pure delay $Delay_{pure}$ respectively – for which separate IO automaton will be constructed. In all cases the same basic technique will be used to construct the IO automata. Each state label consists of a string of values which encode the input history required to compute the next output. For example, for a pure delay of *i* time-units, the input values are reproduced on the output after *i* time-units. Each state label will therefore be a string of length *i*. The next output value will be given by the head of *s*, $hd(s)$, and the next state will be computed by joining the input value *v* onto the rear of the tail of *s*, i.e., $tl(s) \frown [v]$. Having constructed the IO automaton, the special string state labelling can be replaced with the usual natural number labelling, since the labelling itself no longer has further significance.

6.3.2.7.1 Inertial delay An inertial delay $IDelay(arg, const, int)$ produces an initial output value *const*, and then the input values *arg* are reproduced on the output after a delay of *int* time units. However if the input is stable for fewer than *int* time units then the input change is ignored. For example, if $arg = In$, $const = 0$ and $int = 3$ then $Out = IDelay(In, 0, 3)$:

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
In	1	1	0	0	0	1	1	1	0	0	1	0	0	0	0	0	...
Out	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	...

The semantics for inertial delay is given by:

$Delays = IDelay \mid Delay$

$RowOfVal = Val^+$

$$\begin{array}{lrl} IDelay :: & arg & : Expression \\ & const & : RowOfVal \\ & int & : \mathbb{N} \end{array}$$

$MIDelay : IDelay \times Env \to SemObj$

$MIDelay[\![mk\text{-}IDelay(e,c,i)]\!]\sigma \quad \triangle$
  let $exp\text{-}obj = MExpression[\![e]\!]\sigma$ **in**
  let $w =$ **card** $outputs(exp\text{-}obj)$ **in**
  $apply(mk\text{-}idel(c,i,w), exp\text{-}obj)$

where $mk\text{-}idel(c,i,w)$ is a *SemObj*:

$mk\text{-}idel : RowOfVal \times \mathbb{N} \times Nat_1 \to SemObj$

$mk\text{-}idel(c,i,w) \quad \triangle$
  let $states = \{s \frown t \in RowOfVal^* \mid$ **len** $s \frown t = i,$
    elems $s = \{a_1\},$ **elems** $t = \{a_2\}, a_1 \in RowOfVal, a_2 \in RowOfVal\}$ **in**
  let $inits = \{s \in states \mid$ **elems** $s = \{c\}\}$ **in**
  let $ts = \{mk\text{-}Trans(p, mapify(v,-1), mapify($**hd** $p, 1), q) \mid$
    $p \in states, v \in RowOfVal, q = choose\text{-}state(p,v)\}$ **in**
  $mk\text{-}IOA(states, \{-w,\ldots,-1\}, \{1,\ldots,w\}, ts, inits)$

pre len $c = w$

where *choose-state* decides whether the input change should be ignored or not, by calculating the next state:

$choose\text{-}state : RowOfVal^+ \times RowOfVal \to RowOfVal^+$

$choose\text{-}state(s,a) \quad \triangle$ **if** $s($**len** $s) = a$
  then tl $s \frown [a]$
  else let $n = ($**len** $s)$-1 **in**
    $[$**hd** $s]^n \frown [a]$

For example, the IO automaton for the three-unit inertial delay $IDelay(In, x, 3)$, with initial output value $const = 0$ and $i = 3$ will have:

$$\begin{array}{rcl} states & = & \{[0,0,0],[0,0,1],[0,1,1],[1,1,1],[1,0,0],[1,1,0]\} \\ inits & = & \{[0,0]\} \\ ts & = & \{mk\text{-}Trans([0,0,0],\{-1 \mapsto 0\},\{1 \mapsto 0\},[0,0,0]), \\ & & mk\text{-}Trans([0,0,0],\{-1 \mapsto 1\},\{1 \mapsto 0\},[0,0,1]), \end{array}$$

$$
\begin{aligned}
&mk\text{-}Trans([0,0,1],\{-1\mapsto 0\},\{1\mapsto 0\},[0,0,0]),\\
&mk\text{-}Trans([0,0,1],\{-1\mapsto 1\},\{1\mapsto 0\},[0,1,1]),\\
&mk\text{-}Trans([0,1,1],\{-1\mapsto 0\},\{1\mapsto 0\},[0,0,0]),\\
&mk\text{-}Trans([0,1,1],\{-1\mapsto 1\},\{1\mapsto 0\},[1,1,1]),\\
&mk\text{-}Trans([1,1,1],\{-1\mapsto 0\},\{1\mapsto 1\},[1,1,0]),\\
&mk\text{-}Trans([1,1,1],\{-1\mapsto 1\},\{1\mapsto 1\},[1,1,1]),\\
&mk\text{-}Trans([1,0,0],\{-1\mapsto 0\},\{1\mapsto 1\},[0,0,0]),\\
&mk\text{-}Trans([1,0,0],\{-1\mapsto 1\},\{1\mapsto 1\},[1,1,1]),\\
&mk\text{-}Trans([1,1,0],\{-1\mapsto 0\},\{1\mapsto 1\},[1,0,0]),\\
&mk\text{-}Trans([1,1,0],\{-1\mapsto 1\},\{1\mapsto 1\},[1,1,1])\}
\end{aligned}
$$

The corresponding IO automaton for this example appears in Figure 6.6.

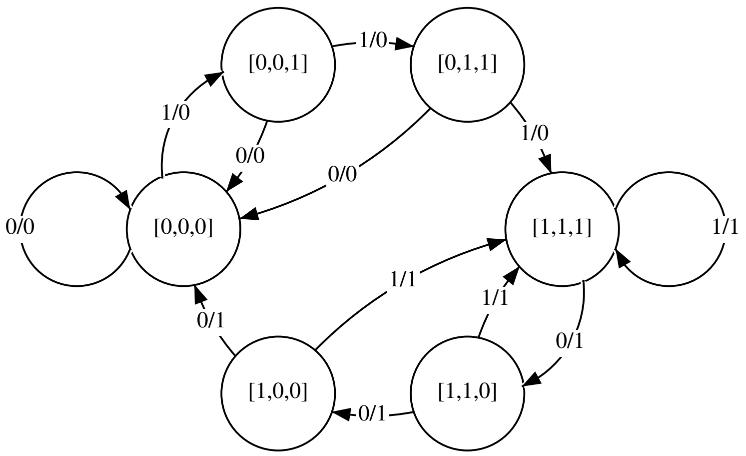

Figure 6.6: Three-Unit Inertial Delay Example IOA

An interesting characteristic of inertial delay is that even though it only requires a finite input history to be stored, in order to calculate its next output value and transition, the full trace history must be available to calculate the last output value, which is of course necessary to calculate the next output.

6.3.2.7.2 Ambiguity Delay Ambiguity delay requires four parameters to describe it: two constant values $const_1$, $const_2 \in Val$ and two delay intervals int_1, int_2.

In [HWM90b], the following informal description for ambiguity delay is given, when considering the function $Delay(const_1, int_1, const_2, int_2)$:

> The initial value of this function is '$const_1$'. If there exists a sequence of int_2 equal input values occurring between the current time and $int_1 + int_2$-1 units of time ago then the output of the function is the input value int_2 units of time ago, otherwise the output is '$const_2$'.

Assume for the purposes of illustration only, that two distinct constants (x, y) are used to represent the values in the value set *Val* assigned to the constants $const_1$ and $const_2$. For example, if $int_1 = 2$ and $int_2 = 5$ then $Out = Delay(In, x, 2, y, 5)$ will have the following effect:

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
In	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	...
Out	x	x	y	y	y	y	y	y	y	0	0	y	y	y	1	1	...

The semantics for ambiguity delay is given by:

$RowOfVal = Val^+$

$$\begin{aligned} Delay ::\quad arg &: Expression \\ const_1 &: RowOfVal \\ int_1 &: \mathbb{N} \\ const_2 &: RowOfVal \\ int_2 &: \mathbb{N} \end{aligned}$$

$MDelay : Delay \times Env \rightarrow SemObj$

$$\begin{aligned} & MDelay[\![mk\text{-}Delay(e, c_1, i_1, c_2, i_2)]\!]\sigma \quad \triangleq \\ & \quad \textbf{let } exp\text{-}obj = MExpression[\![e]\!]\sigma \textbf{ in} \\ & \quad \textbf{let } w = \textbf{card } outputs(exp\text{-}obj) \textbf{ in} \\ & \quad apply(mk\text{-}del(c_1, i_1, c_2, i_2, w), exp\text{-}obj) \end{aligned}$$

where $mk\text{-}del(c_1, i_1, c_2, i_2, w)$ is a *SemObj*:

$mk\text{-}del : RowOfVal \times \mathbb{N} \times RowOfVal \times \mathbb{N} \times Nat_1 \rightarrow SemObj$

$$\begin{aligned} & mk\text{-}del(c_1, i_1, c_2, i_2, w) \quad \triangleq \\ & \quad \textbf{let } states = \{s \in RowOfVal^* \mid \textbf{len } s = i_1 + i_2\text{-}1\} \textbf{ in} \\ & \quad \textbf{let } inits = \{s \in states \mid \textbf{elems } s = \{c_1\}\} \textbf{ in} \\ & \quad \textbf{let } ts = \{mk\text{-}Trans(p, mapify(v, -1), mapify(\textbf{hd } p, 1), \textbf{tl } p \frown [v]) \mid \\ & \qquad\qquad p \in states, q = next\text{-}output(p, i_1, i_2, c_2), v \in RowOfVal\} \textbf{ in} \\ & \quad mk\text{-}IOA(states, \{-w, \ldots, -1\}, \{1, \ldots, w\}, ts, inits) \end{aligned}$$

$\textbf{pre len } c_1 = w \wedge \textbf{len } c_2 = w$

The function *next-output* determines the next output value:

$next\text{-}output : RowOfVal^* \times \mathbb{N} \times \mathbb{N} \times RowOfVal \rightarrow RowOfVal$

$next\text{-}output(s, m, n, c) \quad \triangleq \quad \textbf{if } subsequence(s, n) \textbf{ then } s(m) \textbf{ else } c$

The function $subsequence(s, n)$ checks whether the sequence s contains a sub-sequence of identical values of length $\geq n$:

$subsequence : RowOfVal^* \times \mathbb{N} \rightarrow \mathbb{B}$

$subsequence(s, n) \quad \triangleq \quad \exists s_1, s_2 \in RowOfVal^*, v \in RowOfVal \cdot s = s_1 \frown [v]^n \frown s_2$

As before, the string labelling for each state records the finite input history so that the next transition can be determined, depending on the current input value. For example, the IO automaton for the two-unit ambiguity delay $Delay(In, x, 2, y, 2)$, with constant output values $const_1 = x$, $const_2 = y$, and delay constants $i_1 = 2$, $i_2 = 2$, will have:

$$
\begin{array}{rcl}
states & = & \{[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1], \\
& & [x,x,x],[x,x,0],[x,0,0],[x,x,1],[x,1,1],[x,0,1],[x,1,0]\} \\
inits & = & \{[x,x,x]\} \\
ts & = & \{\textit{mk-Trans}([x,x,x],\{-1 \mapsto 0\},\{1 \mapsto x\},[x,x,0]), \\
& & \textit{mk-Trans}([x,x,0],\{-1 \mapsto 0\},\{1 \mapsto x\},[x,0,0]), \\
& & \textit{mk-Trans}([x,0,0],\{-1 \mapsto 0\},\{1 \mapsto 0\},[0,0,0]), \\
& & \textit{mk-Trans}([x,x,x],\{-1 \mapsto 1\},\{1 \mapsto x\},[x,x,1]), \\
& & \textit{mk-Trans}([x,x,0],\{-1 \mapsto 1\},\{1 \mapsto x\},[x,0,1]), \\
& & \textit{mk-Trans}([x,0,0],\{-1 \mapsto 1\},\{1 \mapsto 0\},[0,0,1]), \\
& & \textit{mk-Trans}([x,x,1],\{-1 \mapsto 0\},\{1 \mapsto x\},[x,1,0]), \\
& & \textit{mk-Trans}([x,1,1],\{-1 \mapsto 0\},\{1 \mapsto 1\},[1,1,0]), \\
& & \textit{mk-Trans}([x,x,1],\{-1 \mapsto 1\},\{1 \mapsto x\},[x,1,1]), \\
& & \textit{mk-Trans}([x,1,1],\{-1 \mapsto 1\},\{1 \mapsto 1\},[1,1,1]), \\
& & \textit{mk-Trans}([x,0,1],\{-1 \mapsto 0\},\{1 \mapsto x\},[0,1,0]), \\
& & \textit{mk-Trans}([x,0,1],\{-1 \mapsto 1\},\{1 \mapsto x\},[0,1,1]), \\
& & \textit{mk-Trans}([x,1,0],\{-1 \mapsto 0\},\{1 \mapsto x\},[1,0,0]), \\
& & \textit{mk-Trans}([x,1,0],\{-1 \mapsto 1\},\{1 \mapsto x\},[1,0,1]), \\
& & \textit{mk-Trans}([0,0,0],\{-1 \mapsto 0\},\{1 \mapsto 0\},[0,0,0]), \\
& & \textit{mk-Trans}([0,0,1],\{-1 \mapsto 0\},\{1 \mapsto 0\},[0,1,0]), \\
& & \textit{mk-Trans}([0,1,0],\{-1 \mapsto 0\},\{1 \mapsto y\},[1,0,0]), \\
& & \textit{mk-Trans}([0,1,1],\{-1 \mapsto 0\},\{1 \mapsto 1\},[1,1,0]), \\
& & \textit{mk-Trans}([1,0,0],\{-1 \mapsto 0\},\{1 \mapsto 0\},[0,0,0]), \\
& & \textit{mk-Trans}([1,0,1],\{-1 \mapsto 0\},\{1 \mapsto y\},[0,1,0]), \\
& & \textit{mk-Trans}([1,1,0],\{-1 \mapsto 0\},\{1 \mapsto 1\},[1,0,0]), \\
& & \textit{mk-Trans}([1,1,1],\{-1 \mapsto 0\},\{1 \mapsto 1\},[1,1,0]), \\
& & \textit{mk-Trans}([0,0,0],\{-1 \mapsto 1\},\{1 \mapsto 0\},[0,0,1]), \\
& & \textit{mk-Trans}([0,0,1],\{-1 \mapsto 1\},\{1 \mapsto 0\},[0,1,1]), \\
& & \textit{mk-Trans}([0,1,0],\{-1 \mapsto 1\},\{1 \mapsto y\},[1,0,1]), \\
& & \textit{mk-Trans}([0,1,1],\{-1 \mapsto 1\},\{1 \mapsto 1\},[1,1,1]), \\
& & \textit{mk-Trans}([1,0,0],\{-1 \mapsto 1\},\{1 \mapsto 0\},[0,0,1]), \\
& & \textit{mk-Trans}([1,0,1],\{-1 \mapsto 1\},\{1 \mapsto y\},[0,1,1]), \\
& & \textit{mk-Trans}([1,1,0],\{-1 \mapsto 1\},\{1 \mapsto 1\},[1,0,1]), \\
& & \textit{mk-Trans}([1,1,1],\{-1 \mapsto 1\},\{1 \mapsto 1\},[1,1,1])\}
\end{array}
$$

The corresponding IO automaton for this example appears in Figure 6.7.

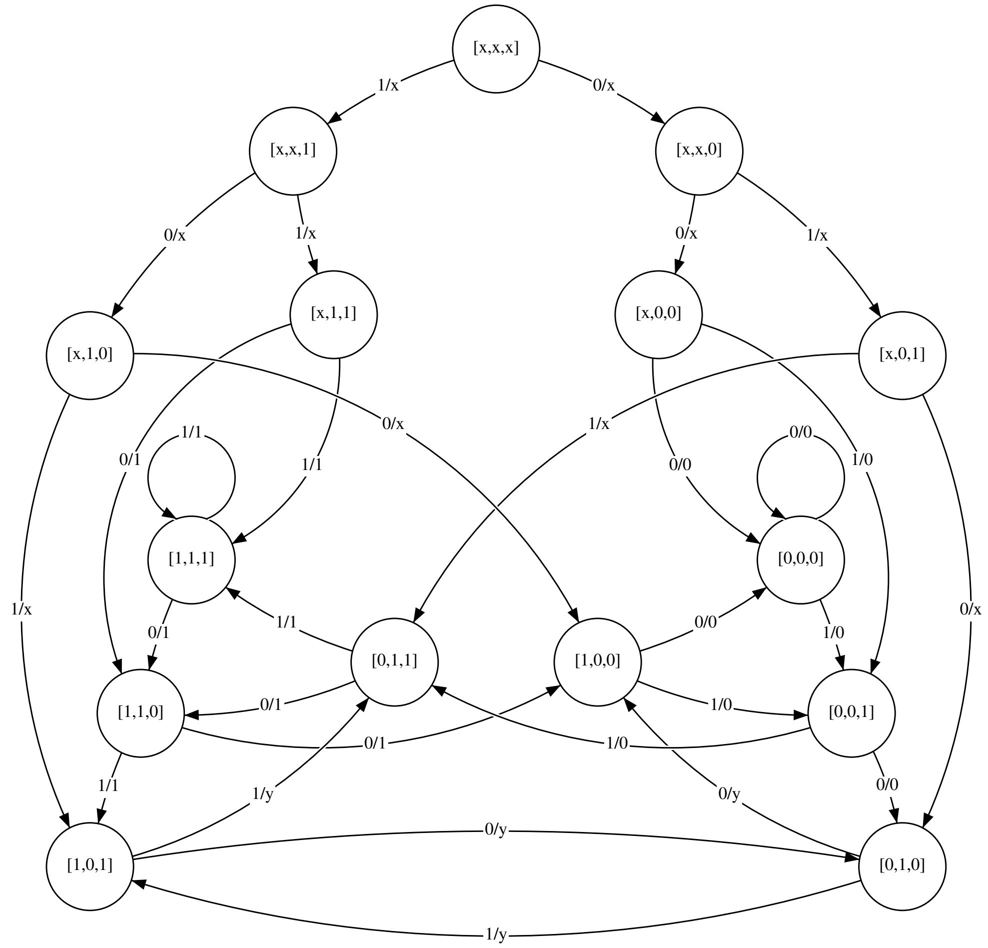

Figure 6.7: Two-Unit Ambiguity Delay Example IOA

This example has been chosen, with $i_1 = i_2 = 2$, so that if y is assigned the same value as x, then the output from the ambiguity delay automaton is equivalent to that from a two-unit transport delay automaton, which will be used as the example in §6.3.2.7.3.

6.3.2.7.3 Pure delay $Delay_{pure}$ Pure delay[HWM90b] is a special case of inertial delay, not explicitly defined in ELLA. For $Delay_{pure}(arg, const, int)$, the initial output value is *const*, and input values of *arg* are reproduced on the output after *int* time units. For example, if $arg = In,\ const = 0,\ int = 2$ and $Val = \{0,1\}$ then $Out = Delay_{pure}(In, 0, 2)$ will produce:

Time	0	1	2	3	4	5	6	7	8	9	10	...
In	1	0	1	1	0	0	1	0	0	0	0	...
Out	0	0	1	0	1	1	0	0	1	0	0	...

Pure delay has not been separately defined in Core ELLA, but can be written in terms of inertial delay. For example:

$$Delay_{pure}(e,c,3) = IDelay(IDelay(IDelay(e,c,1),c,1),c,1)$$

However, a separate IO automaton representing pure delay will be constructed. It will be trace-equivalent to the IO automaton resulting from the composition of the sequence of unit-delay *IDelay* operations.

The semantics is given by:

$$RowOfVal = Val^{+}$$

$$\begin{array}{lrcl} Delay_{pure} :: & arg & : & Expression \\ & const & : & RowOfVal \\ & int & : & \mathbb{N} \end{array}$$

$$MDelay_{pure} : Delay_{pure} \times Env \to SemObj$$

$$\begin{array}{l} MDelay_{pure}[\![mk\text{-}Delay_{pure}(e,c,i)]\!]\sigma \quad \triangleq \\ \quad \mathbf{let}\ exp\text{-}obj = MExpression[\![e]\!]\sigma\ \ \mathbf{in} \\ \quad \mathbf{let}\ w = \mathbf{card}\ outputs(exp\text{-}obj)\ \ \mathbf{in} \\ \quad apply(mk\text{-}del_{pure}(c,i,w), exp\text{-}obj) \end{array}$$

where $mk\text{-}del_{pure}(c,i,w)$ is a *SemObj*:

$$mk\text{-}del_{pure} : RowOfVal \times \mathbb{N} \times Nat_1 \to SemObj$$

$$\begin{array}{l} mk\text{-}del_{pure}(c,i,w) \quad \triangleq \\ \quad \mathbf{let}\ states = \{s \in RowOfVal^{*} \mid \mathbf{len}\, s = i\}\ \ \mathbf{in} \\ \quad \mathbf{let}\ inits = \{s \in states \mid \mathbf{elems}\, s = \{c\}\}\ \ \mathbf{in} \\ \quad \mathbf{let}\ ts = \{mk\text{-}Trans(p, mapify(v,-1), mapify(\mathbf{hd}\, p, 1), (\mathbf{tl}\, p) \curvearrowright [v]) \mid \\ \qquad\qquad p \in states, v \in RowOfVal\}\ \ \mathbf{in} \\ \quad mk\text{-}IOA(states, \{-w,\ldots,-1\}, \{1,\ldots,w\}, ts, inits) \end{array}$$

$$\mathbf{pre}\ \mathbf{len}\, c = w$$

For example, the IO automaton for the two-unit delay $Delay_{pure}(In,0,2)$, with initial output value $const = 0$ and $i = 2$ will have:

$$\begin{array}{rcl} states & = & \{[0,0],[0,1],[1,0],[1,1]\} \\ inits & = & \{[0,0]\} \\ ts & = & \{mk\text{-}Trans([0,0],\{-1 \mapsto 0\},\{1 \mapsto 0\},[0,0]), \\ & & \ mk\text{-}Trans([0,0],\{-1 \mapsto 1\},\{1 \mapsto 0\},[0,1]), \\ & & \ mk\text{-}Trans([0,1],\{-1 \mapsto 0\},\{1 \mapsto 0\},[1,0]), \\ & & \ mk\text{-}Trans([0,1],\{-1 \mapsto 1\},\{1 \mapsto 0\},[1,1]), \\ & & \ mk\text{-}Trans([1,0],\{-1 \mapsto 0\},\{1 \mapsto 1\},[0,0]), \\ & & \ mk\text{-}Trans([1,0],\{-1 \mapsto 1\},\{1 \mapsto 1\},[0,1]), \\ & & \ mk\text{-}Trans([1,1],\{-1 \mapsto 0\},\{1 \mapsto 1\},[1,0]), \\ & & \ mk\text{-}Trans([1,1],\{-1 \mapsto 1\},\{1 \mapsto 1\},[1,1])\} \end{array}$$

The corresponding IO automaton for this example appears in Figure 6.8.

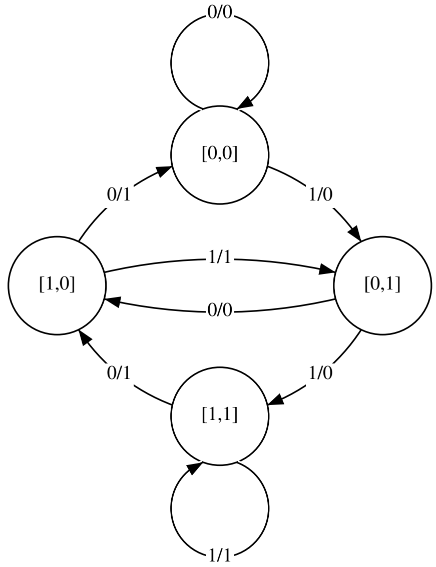

Figure 6.8: Two-Unit Pure Delay Example IOA

6.3.2.7.4 Transport Delay *trans* Transport delay can be considered as a special case of ambiguity delay $Delay(arg, const1, int1, const2, int2)$ where the constant values $const1, const2$ and the delay intervals $int1, int2$ are both equal; it is not explicitly defined in ELLA. For $Delay_{trans}(arg, const, int)$ the initial output values are given by *const*. Then the input value *arg* appears on the output after *int* time units, unless the input is stable for less than *int* units, in which case the output is replaced by *const*. For an example, assume again that there is a third distinct value $x \in Val$ to which *const* can be set and that $int = 2$, then $Out = Delay_{trans}(In, x, 2)$ will have the following effect:

Time	0	1	2	3	4	5	6	7	8	9	10	...
In	0	1	1	1	0	0	1	0	0	0	0	...
Out	x	x	x	1	1	1	0	0	x	0	0	...

A new IO automaton will now be constructed to represent transport delay:

$$RowOfVal = Val^{+}$$

$$\begin{aligned} Delay_{trans} :: \quad arg &: Expression \\ const &: RowOfVal \\ int &: \mathbb{N} \end{aligned}$$

$$MDelay_{trans} : Delay_{trans} \times Env \to SemObj$$

$$MDelay_{trans}[\![mk\text{-}Delay_{trans}(e, c, i)]\!]\sigma \quad \triangleq$$
$$\textbf{let } exp\text{-}obj = MExpression[\![e]\!]\sigma \textbf{ in}$$
$$\textbf{let } w = \textbf{card } outputs(exp\text{-}obj) \textbf{ in}$$
$$apply(mk\text{-}del_{trans}(c, i, w), exp\text{-}obj)$$

where $mk\text{-}del_{trans}(c,i,w)$ is given by:

$$mk\text{-}del_{trans} : RowOfVal \times \mathbb{N} \times \mathbb{N}_1 \rightarrow SemObj$$

$$\begin{array}{l} mk\text{-}del_{trans}(c,i,w) \quad \triangleq \\ \quad \textbf{let } states = \{s \mid s = [f]^k \frown [g]^l \frown [h]^m \wedge \textbf{len } s = i \wedge f,g,h \in RowOfVal \wedge k,l,m \in \mathbb{N}\} \ \textbf{ in} \\ \quad \textbf{let } inits = \{s \in states \mid \textbf{elems } s = \{c\}\} \ \textbf{ in} \\ \quad \textbf{let } ts = \{mk\text{-}Trans(p, mapify(v,-1), mapify(\textbf{hd } p, 1), q) \mid \\ \qquad\qquad p \in states, v \in RowOfVal, q = next\text{-}state(p,v,c,i)\} \ \textbf{ in} \\ \quad mk\text{-}IOA(states, \{-w,\ldots,-1\}, \{1,\ldots,w\}, ts, inits) \end{array}$$

The function *next-state* determines, from the current state label, whether the input has remained stable long enough so that a change in output should still take place; if the last element in the label is different from the input value, then a change is taking place and the label sequence is reconstructed:

$$next\text{-}state : RowOfVal^* \times RowOfVal \times \mathbb{N} \rightarrow RowOfVal^*$$

$$\begin{array}{ll} next\text{-}state(p,v,c,i) \ \triangleq & \textbf{if } p(\textbf{len } p) = v \\ & \textbf{then tl } p \frown [v] \\ & \textbf{else let } a = \textbf{hd } p \ \textbf{ in} \\ & \quad \textbf{let } j = \iota j \in \mathbb{N} \cdot [a]^j \frown s = p \wedge \textbf{hd } s \neq a \ \textbf{ in} \\ & \quad [a]^{j\text{-}1} \frown [c]^{i\text{-}j} \frown [v] \end{array}$$

For example, consider the state-label 0011 for $i = 4$. If the next input is 1 then the next state will be 0111, with output 0. However if the next input is 0 then the next state will be $0cc0$, again with output 0.

The IO automaton for the two-unit transport delay $Delay_{trans}(In, x, 2)$, with initial output value $const = x$ and $i = 2$ will have:

$$\begin{array}{rcl} states & = & \{[0,0],[0,1],[1,0],[1,1], \\ & & [x,x],[x,0],[x,1]\} \\ inits & = & \{[x,x]\} \\ ts & = & \{mk\text{-}Trans([0,0],\{1 \mapsto 0\},\{1 \mapsto 0\},[0,0]), \\ & & mk\text{-}Trans([0,0],\{1 \mapsto 1\},\{1 \mapsto 0\},[0,1]), \\ & & mk\text{-}Trans([0,1],\{1 \mapsto 0\},\{1 \mapsto 0\},[x,0]), \\ & & mk\text{-}Trans([0,1],\{1 \mapsto 1\},\{1 \mapsto 0\},[1,1]), \\ & & mk\text{-}Trans([1,0],\{1 \mapsto 0\},\{1 \mapsto 1\},[0,0]), \\ & & mk\text{-}Trans([1,0],\{1 \mapsto 1\},\{1 \mapsto 1\},[x,1]), \\ & & mk\text{-}Trans([1,1],\{1 \mapsto 0\},\{1 \mapsto 1\},[1,0]), \\ & & mk\text{-}Trans([1,1],\{1 \mapsto 1\},\{1 \mapsto 1\},[1,1]), \\ & & mk\text{-}Trans([x,x],\{1 \mapsto 0\},\{1 \mapsto x\},[x,0]), \\ & & mk\text{-}Trans([x,x],\{1 \mapsto 1\},\{1 \mapsto x\},[x,1]), \\ & & mk\text{-}Trans([x,0],\{1 \mapsto 0\},\{1 \mapsto x\},[0,0]), \\ & & mk\text{-}Trans([x,0],\{1 \mapsto 1\},\{1 \mapsto x\},[x,1]), \\ & & mk\text{-}Trans([x,1],\{1 \mapsto 0\},\{1 \mapsto x\},[x,0]), \\ & & mk\text{-}Trans([x,1],\{1 \mapsto 1\},\{1 \mapsto x\},[1,1])\} \end{array}$$

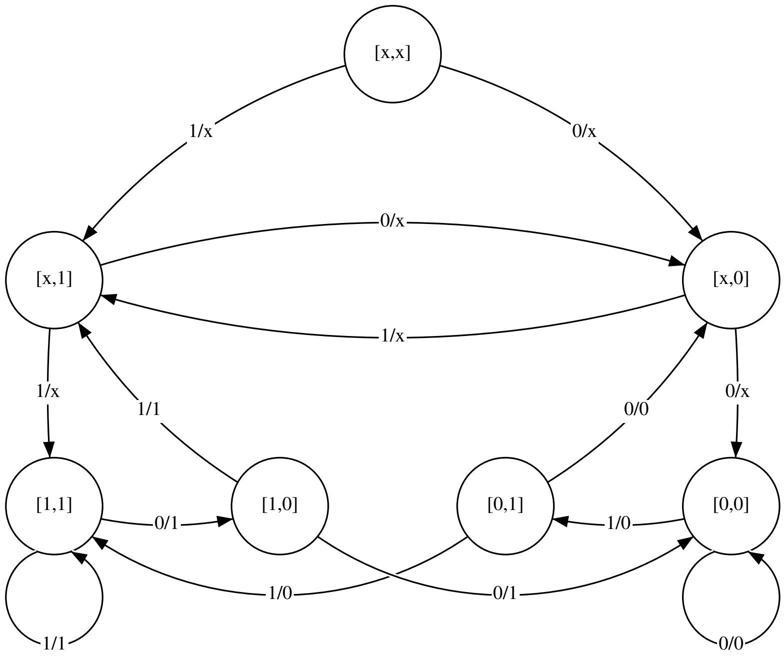

Figure 6.9: Two-Unit Transport Delay Example IOA

The corresponding IO automaton for this example appears in Figure 6.9.

This automaton will be trace-equivalent to the IO automaton given in §6.3.2.7.2 for $Delay(In, x, 2, x, 2)$, having replaced variables y by x. If the ambiguity delay automaton for $Delay(In, x, 2, x, 2)$ is minimised, by merging equivalent nodes are removing duplicate edges, then an automaton equivalent to Figure 6.9 is produced, which is shown in Figure 6.10.

6.3.2.8 Sample

A sample operation is included in Core ELLA, and will be part of future versions of ELLA, so that elementary retiming operations can be applied to design modules. One part of the design can then be considered to be running at a faster or slower clock rate, in terms of the ELLA time-unit, than other parts in the same interacting environment. The sample operation $Sample(expr, const, skew, int)$ selects the input value given by *expr* at regular intervals *int* and outputs this value for the duration of the interval. The start of the sampling process can also be skewed by an amount *skew*, where an initial value *const* is produced until the sampling process begins at time *skew*. Fresh input samples will be taken at times $(skew + int)$, $(skew + 2 \times int)$, and so on. For example, if $expr = In$, $const = x$, $skew = 2$ and $int = 4$, then $Out = Sample(In, x, 2, 4)$ is given by:

Time	0	1	2	3	4	5	6	7	8	9	10	...
In	1	0	0	0	1	0	1	0	0	0	0	...
Out	x	x	0	0	0	0	1	1	1	1	0	...

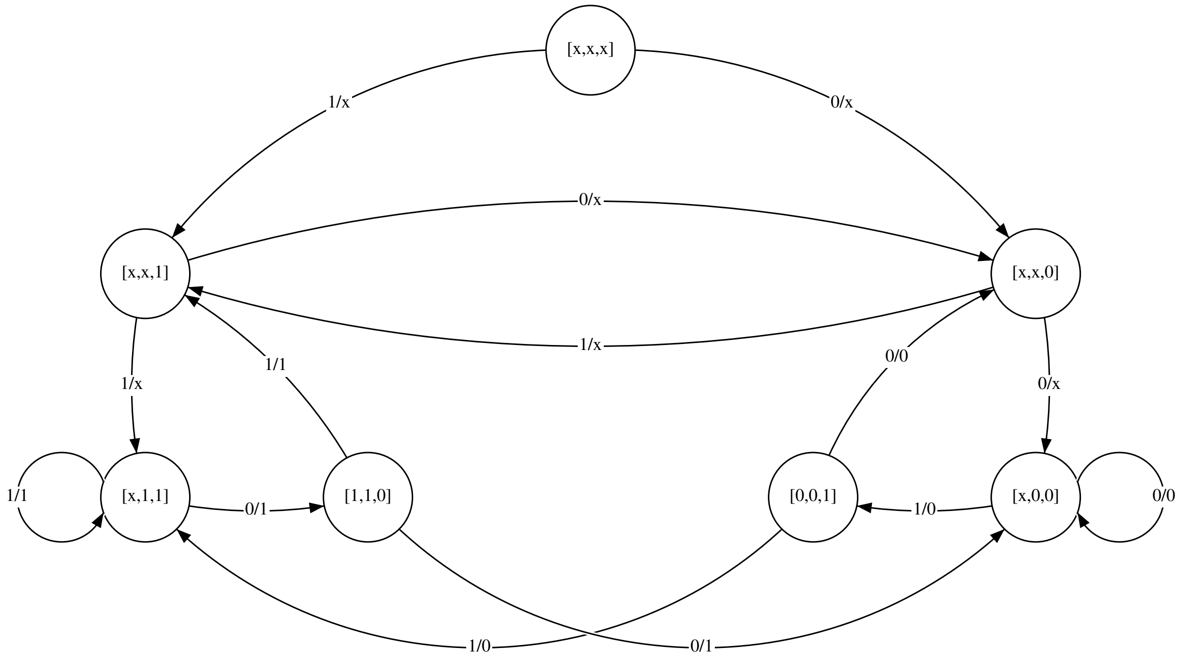

Figure 6.10: Modified Two-Unit Ambiguity Delay Example IOA

The semantics for *Sample* is derived in a similar way to that of delays, by applying a semantic object, *Sampler* to the argument *expr*. The states are constructed as follows: there is an initial state, $-skew$, followed by a sequence of states $-skew+1, -skew+2, \ldots, -1$ where the output is *const* regardless of the value on the input. Then for each possible input vector *v*, a cycle of states and transitions is formed beginning and ending in state 0. The cycle has *n* transitions for a sampling interval *n*, The intermediate states are labelled (for construction purposes only) by the pair (v, i) where $i \in \{1, \ldots, n-1\}$ is the index position on the cycle. For each state (v,j) there is a single transition to state $(v, j+1), j \in \{1, \ldots, n\text{-}2\}$. The transitions exiting from state 0 are both input- and output-labelled (v/v), whereas all other transitions in the cycle are just output-labelled $(/v)$, because the input value does not affect the output value.

$$\begin{aligned} \textit{Sample} \;::\; \textit{expr} &: \textit{Expression} \\ \textit{const} &: \textit{Val}^{+} \\ \textit{skew} &: \mathbb{N} \\ \textit{int} &: \mathbb{N}_1 \end{aligned}$$

$$\textit{MSample} : \textit{Expression} \times \textit{Val}^{+} \times \mathbb{N} \times \mathbb{N}_1 \rightarrow \textit{SemObj}$$

$\textit{MSample}[\![\textit{expr}, \textit{const}, \textit{skew}, \textit{int}]\!]\sigma \quad \triangleq$

let $\textit{exp-obj} = \textit{MExpression}[\![\textit{expr}]\!]\sigma$ **in**

let $\textit{width} =$ **card** $\textit{outputs}(\textit{exp-obj})$ **in**

$\textit{apply}(\textit{Sampler}(\textit{const}, \textit{skew}, \textit{int}, \textit{width}), \textit{exp-obj})$

$Sampler : Val^{+} \times \mathbb{N} \times \mathbb{N}_1 \times \mathbb{N}_1 \to SemObj$

$$
\begin{array}{l}
Sampler(c,s,n,w) \quad \triangleq \\
\quad \textbf{let } inits = \{-s\} \textbf{ in} \\
\quad \textbf{let } skews = \{-s+i \mid i \in \{1,\ldots,s\}\} \textbf{ in} \\
\quad \textbf{let } cyclers = \{(v,i) \mid v \in Val^{w} \wedge i \in \{2,\ldots,n-1\}\} \textbf{ in} \\
\quad \textbf{let } states = inits \cup skews \cup cyclers \textbf{ in} \\
\quad \textbf{let } ts = \{mk\text{-}Trans(-s+j,\{\},mapify(c,1),-s+j+1) \mid j \in \{0,\ldots,s-1\}\} \cup \\
\qquad\qquad \{mk\text{-}Trans(0,mapify(v,-1),mapify(v,1),0) \mid v \in Val^{w} \wedge n = 1\} \cup \\
\qquad\qquad \{mk\text{-}Trans(0,mapify(v,-1),mapify(v,1),(v,1)), \\
\qquad\qquad\; mk\text{-}Trans((v,n-1),\{\},mapify(v,1),0) \mid v \in Val^{w} \wedge n > 1\} \cup \\
\qquad\qquad \{mk\text{-}Trans((v,i),\{\},mapify(v,1),(v,i+1)) \\
\qquad\qquad\qquad \mid v \in Val^{w} \wedge i \in \{1,\ldots,n-2\}\} \textbf{ in} \\
\quad mk\text{-}IOA(states,\{-w,\ldots,-1\},\{1,\ldots,w\},ts,inits)
\end{array}
$$

$\textbf{pre len } c = w$

For example, the function object for the sample operation $Sample(In,x,2,4)$, with $Val = \{0,1\}$, is given by:

$$
\begin{array}{rcl}
states & = & \{-2,-1,0,(0,1),(0,2),(0,3),(1,1),(1,2),(1,3)\} \\
inits & = & \{-2\} \\
ts & = & \{mk\text{-}Trans(-2,\{\},\{1 \mapsto x\},-1), \\
 & & \;mk\text{-}Trans(-1,\{\},\{1 \mapsto x\},0), \\
 & & \;mk\text{-}Trans(0,\{-1 \mapsto 0\},\{1 \mapsto 0\},(0,1)), \\
 & & \;mk\text{-}Trans(0,\{-1 \mapsto 1\},\{1 \mapsto 1\},(1,1)), \\
 & & \;mk\text{-}Trans((0,1),\{\},\{1 \mapsto 0\},(0,2)), \\
 & & \;mk\text{-}Trans((0,2),\{\},\{1 \mapsto 0\},(0,3)), \\
 & & \;mk\text{-}Trans((0,3),\{\},\{1 \mapsto 0\},0), \\
 & & \;mk\text{-}Trans((1,1),\{\},\{1 \mapsto 1\},(1,2)), \\
 & & \;mk\text{-}Trans((1,2),\{\},\{1 \mapsto 1\},(1,3)), \\
 & & \;mk\text{-}Trans((1,3),\{\},\{1 \mapsto 1\},0)\}
\end{array}
$$

Having constructed the automaton, the special state labelling can be replaced with the usual natural number labelling, since the labelling itself no longer has further significance.

The corresponding IO automaton for this example appears in Figure 6.11.

6.3.2.9 BIOPS: Built-in Operators

Unary Operators *BKE* has one built-in unary operator, the negation operator. It is simply applied to the result of its operand expression.

$Not \;::\; negated \;:\; Expression$

$MNot : Not \to Env \to SemObj$

$MNot[\![mk\text{-}Not(n)]\!]\sigma \quad \triangleq \quad apply(not\text{-}obj, MExpression[\![n]\!]\sigma)$

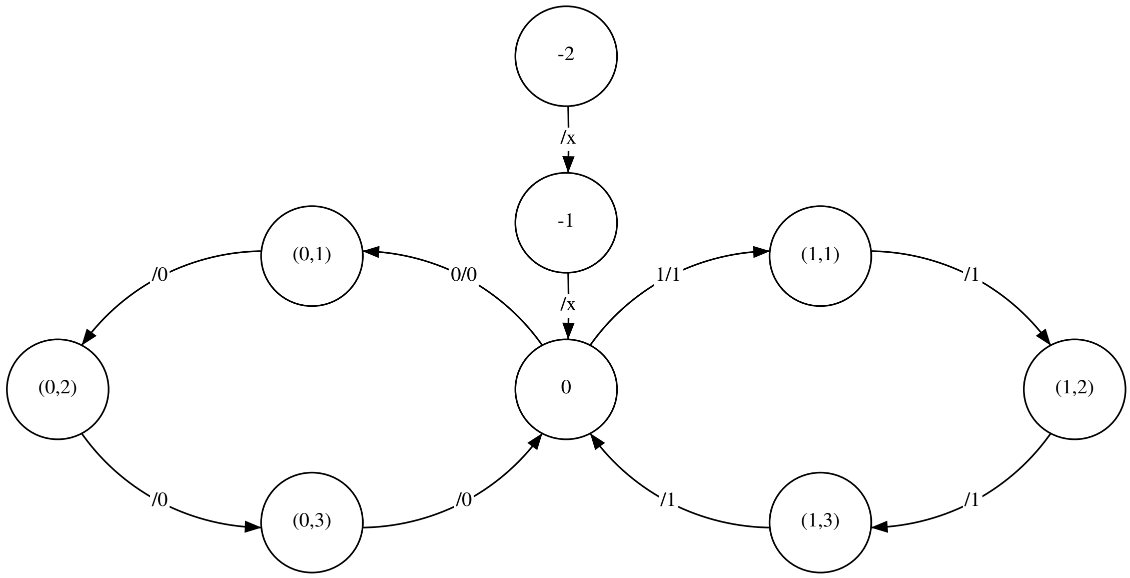

Figure 6.11: IO Automaton for Sampling Every 4 Steps with 2-unit Skew

where *not-obj* is a predefined *SemObj* given by

$$\textit{not-obj} = \textit{mk-IOA}(\{0\},\{-1\},\{1\},ts,\{0\})$$

and

$$ts = \{\textit{mk-Trans}(0,\{-1 \mapsto v\},\{1 \mapsto \neg v\},0) \mid v \in \textit{Val}\}$$

Binary operators Binary operations such as OR are treated in a similar way to negation, by applying predefined function objects.

$$\textit{Binop} = \text{OR} \mid \text{AND}$$

$$\begin{array}{lrcl} \textit{Infix} :: & \textit{arg}1 & : & \textit{Expression} \\ & \textit{op} & : & \textit{Binop} \\ & \textit{arg}2 & : & \textit{Expression} \end{array}$$

$$\textit{MInfix} : \textit{Infix} \to \textit{Env} \to \textit{SemObj}$$

$$\begin{array}{l} \textit{MInfix}[\![\textit{mk-InFix}(e_1,o,e_2)]\!]\sigma \quad \triangleq \\ \quad \textit{apply}(\textit{MBinop}[\![o]\!],[\textit{MExpression}[\![e_1]\!]\sigma,\textit{MExpression}[\![e_2]\!]\sigma]) \end{array}$$

The function *MBinop* can be defined on a case-by-case basis for as many binary operators as we wish to include as primitives in the language. For example:

$$\textit{MBinop}(\text{OR}) = \textit{mk-IOA}(\{0\},\{-1,-2\},\{1\},ts,\{0\})$$

where

$$ts = \{\textit{mk-Trans}(0,\{-1 \mapsto v_1, -2 \mapsto v_2\},\{1 \mapsto v_1 \vee v_2\},0) \mid v_1,v_2 \in \textit{Val}\}$$

and $\vee$ is the disjunction operator for the underlying data type *Val*.

6.3.3 Function declarations

To evaluate a function definition we first evaluate the function body in the given environment. (Aside—Note that if the expression defining the function body does not contain a free reference to some formal parameter, the resulting automaton will not have the corresponding name in its input set. When we apply the function, the result is correct since the missing inputs do not impose any conditions on the transition set). We then rename the inputs corresponding to the formal parameters to the appropriate internal names, i.e. negative numbers. Finally, we bind the function name to a *SemObj* consisting of a count of the parameters and the automaton we have just constructed.

The renaming mapping for the automaton is produced by the function *make-subst*, which has to take account of the fact that the formal parameters may not all appear in the input set. Our convention for composition of maps is that $f \circ g$ means "apply f to the result of g"—that is, "do g then f". Note the precondition on the arguments to *number-inputs*, which ensures that the new numerical names do not conflict with existing input names.

$$
\begin{aligned}
&\mathit{FnDec} :: \quad \mathit{name} : \mathit{Id} \\
&\qquad\quad \mathit{params} : \mathit{Id}^* \\
&\qquad\quad \mathit{body} : \mathit{Expression}
\end{aligned}
$$

$$\mathit{MFnDec} : \mathit{FnDec} \to \mathit{Env} \to \mathit{Env}$$

$$
\begin{aligned}
&\mathit{MFnDec}[\![\textit{mk-FnDec}(n, ps, b)]\!]\sigma \quad \triangleq \\
&\qquad \textbf{let } \textit{obj-body} = \textit{number-inputs}(\mathit{MExpression}[\![b]\!]\sigma, ps) \textbf{ in} \\
&\qquad \sigma \dagger \{f \mapsto \textit{obj-body}\}
\end{aligned}
$$

$$\textit{number-inputs} : \mathit{IOA} \times \mathit{Id}^* \to \mathit{IOA}$$

$$
\begin{aligned}
&\textit{number-inputs}(a, ids) \quad \triangleq \\
&\qquad \textbf{let } \textit{mk-IOA}(ss, i, o, ts, start) = a \textbf{ in} \\
&\qquad \textbf{let } subst = \textit{make-subst}(i, ids) \textbf{ in} \\
&\qquad \textbf{let } \textit{new-ts} = \{\mu(t, ivals \mapsto ivals(t) \circ subst) \mid t \in ts\} \textbf{ in} \\
&\qquad \textit{mk-IOA}(ss, \textbf{dom}\, subst, o, \textit{new-ts}, start)
\end{aligned}
$$

$$\textbf{pre } inputs(a) \subseteq \mathit{Id}$$

$$\textit{make-subst} : \mathit{Id}\textbf{-set} \times \mathit{Id}^* \to (\mathit{Name} \xleftrightarrow{m} \mathit{Name})$$

$$
\begin{aligned}
&\textit{make-subst}(iset, ilist) \quad \triangleq \\
&\qquad \textbf{let } inv = \{x \mapsto x \mid x \in iset\} \dagger \{ilist(k) \mapsto -k \mid ilist(k) \in iset \wedge k \in \textbf{inds}\, ilist\} \textbf{ in} \\
&\qquad inv^{-1}
\end{aligned}
$$

6.3.4 Closures

A program in *BKE*, or *Closure*, consists of a sequence of function declarations. The meaning of a *Closure* is thus taken as the function environment.

$$\mathit{Closure} = \mathit{FnDec}^+$$

$$MClosure : Closure \rightarrow Env$$

$$MClosure[\![ds]\!] \quad \triangleq \quad MFnDecList[\![ds]\!]$$

$$MFnDecList : FnDec^{*} \rightarrow Env$$

$$MFnDecList[\![ds]\!] \quad \triangleq \quad \begin{array}{l} \textbf{cases}\ ds\ \textbf{of} \\ [\,] \qquad\quad \rightarrow \{\,\} \\ l \frown [dec] \rightarrow MFnDec[\![dec]\!](MFnDecList[\![l]\!]) \\ \textbf{end} \end{array}$$

Chapter 7

History Semantics for Boolean Kernel ELLA

This chapter contains a discussion of history semantics for LD, Logic with Delay, which could form the basis for a more comprehensive history based semantics for *BKE*. A discussion of the issues involved in extending history semantics towards *BKE* is also given.

Further work is needed to develop the history semantics further.

7.1 Introduction

The operational semantics approach constructs a syntax-directed description of the execution process for a language. It also provides an *extension* of the experimental simulation-by-testing techniques often used to validate hardware designs. Consequently, support systems based upon operational semantics can provide simulation facilities as an integral part of the verification system. The fact that such facilities are constructed directly from an agreed semantics enables it to act as a *reference simulator* against which the output of other, more efficient and faster simulators can be compared. In general, simulation facilities are useful to help explore the behaviour when, for example, the verification indicates discrepancies between design and implementation. We therefore suggest that formal techniques do not render testing procedures obsolescent or unnecessary. Instead, the application of formal methods should help make testing procedures more focused and systematic and so less of the gamble they are at present.

7.2 Simplified History Semantics and LD

A natural way of formulating a semantics for ELLA and related languages, like *BKE*, is to exploit a direct model of delay defined in terms of finite histories. Our investigation into this approach began with a study of a very simple mini-language called LD (for Logic+Delay). This provided a useful vehicle for presenting the main ideas behind this semantic approach. Accordingly, we shall introduce the LD language and give history semantics for it in the following sections. Finally, we shall then indicate how this semantics for LD may be extended towards a similar style semantics for *BKE*.

7.3 Boolean Expressions

To introduce the notation and techniques, we give the syntax and semantics of the class of Boolean expressions used in LD. We use a form of syntax that we call a *composite* syntax[1]; it defines the *concrete* syntax of a language, in terms of sets of strings, and also defines an *abstract* syntax for the same language, in terms of sets of tree structures.

Expressions, $(e, e_1, e_2 \in \mathit{Expr})$ given by:-

$$e \quad ::= \quad v \mid \textbf{true} \mid \textbf{false} \mid \overline{e_1} \mid e_1 + e_2$$

An expression can either be a variable $v \in \mathit{Var}$, a lexically determined set of atomic names representing signals, one of the the elements **true** or **false**, the negation of an expression or the logical disjunction of two expressions[2]. The information given above does not determine certain properties needed for parsing, such as the syntactic *associativity* or *precedence* of binary and unary operators. We can safely assume herein that such choices could be effectively made, if ever it became necessary to do so.

The semantics is now given in terms of a *binary relation*, defined by using inference rules. Before we can define this relation we need some further structure.

Basic syntactic classes

$$\mathit{Var} \quad = \quad \{x_0, x_1, x_2, \cdots\} \quad \text{(ranged over by v)}$$

Operational classes

$$\begin{aligned} \mathbb{B} \quad &= \quad \{0,1\} && \text{(ranged over by b)} \\ \mathit{State} \quad &= \quad \mathit{Var} \xrightarrow{m} \mathbb{B} && \text{(ranged over by s)} \end{aligned}$$

where the class $\mathbb{B}$ is comprised of a two element type representing the logic states. The finite, partial mapping (or *environment*), *State*, gives an instantaneous snapshot of the values associated with variables that may occur in expressions. Our operational semantics is expressed using a transition relation, mapping a state and an expression to a value result:

$$(\ \_ \vdash \_ \Rightarrow \_\ ) \subseteq (\mathit{State} \times \mathit{Expr} \times \mathbb{B})$$

where $(\ \sigma \vdash e \Rightarrow b\ )$ means that, given the state σ, the expression e is evaluated to produce value b. The relation $(\ \_ \vdash \_ \Rightarrow \_\ )$ is defined to be the least relation satisfying the rules given in Figure 7.1. In the next section, we use the above when defining the first of the design languages to be studied.

7.4 Logic+Delay language

We now introduce a language, called Logic+Delay (or LD), that uses explicit delay instead of (clocked) latching to provide the memory function. Intuitively, this contains the primitive ingredients for a semantic study of the ELLA language. Accordingly, the Logic+Delay language studied here shares many of the fundamental assumptions made within the full ELLA language[Com90a, Det86, MPT85b].

[1] Some authors call this the abstract syntax (c.f: Plotkin, Hennessy)

[2] We could have included the ‘and’ primitive, either together with or instead of, the ‘or’ primitive ($\_+\_$) which is adopted here. It is immaterial which is used as long as at least one such binary logical operator is included. For either of these choices, the corresponding semantics can easily be defined.

$$\boxed{\text{E:1a}}\;\frac{}{\sigma \vdash \textbf{true} \Rightarrow 1} \qquad \boxed{\text{E:1b}}\;\frac{}{\sigma \vdash \textbf{false} \Rightarrow 0}$$

$$\boxed{\text{E:2}}\;\frac{\sigma(v) = b}{\sigma \vdash v \Rightarrow b}$$

$$\boxed{\text{E:3a}}\;\frac{\sigma \vdash e \Rightarrow 0}{\sigma \vdash \neg e \Rightarrow 1} \qquad \boxed{\text{E:3b}}\;\frac{\sigma \vdash e \Rightarrow 1}{\sigma \vdash \neg e \Rightarrow 0}$$

$$\boxed{\text{E:4a}}\;\frac{\sigma \vdash e_1 \Rightarrow 1}{\sigma \vdash e_1 + e_2 \Rightarrow 1} \qquad \boxed{\text{E:4b}}\;\frac{\sigma \vdash e_2 \Rightarrow 1}{\sigma \vdash e_1 + e_2 \Rightarrow 1}$$

$$\boxed{\text{E:4c}}\;\frac{\sigma \vdash e_1 \Rightarrow 0\,, \qquad \sigma \vdash e_2 \Rightarrow 0}{\sigma \vdash e_1 + e_2 \Rightarrow 0}$$

Figure 7.1: Semantic rules for Boolean Expressions

$$\boxed{\text{E:5a}}\ \frac{\sigma \vdash e_0 \Rightarrow 1\ , \qquad \sigma \vdash e_1 \Rightarrow b}{\sigma \vdash \ \mathsf{case}\ e_0\ \mathsf{of}\ \mathbf{true} : e_1, \mathbf{false} : e_2 \Rightarrow b}$$

$$\boxed{\text{E:5b}}\ \frac{\sigma \vdash e_0 \Rightarrow 0\ , \qquad \sigma \vdash e_2 \Rightarrow b}{\sigma \vdash \ \mathsf{case}\ e_0\ \mathsf{of}\ \mathbf{true} : e_1, \mathbf{false} : e_2 \Rightarrow b}$$

Figure 7.2: The case statement rules

Our presentation of the semantics of LD uses the earlier treatment of Boolean expressions. However, we first extend our expression language by including a basic form of the more extensive 'case' statement used in ELLA. The case statement we have chosen only applies to two-valued types, so essentially providing a conditional construct. It takes the following syntactic form:

$$\mathsf{case}\ e_0\ \mathsf{of}\ \mathbf{true} : e_1, \mathbf{false} : e_2$$

The expression e_0 is known as the *chooser* expression. The syntax of expressions used in LD becomes:

Expressions, $(e \in Expr)$ given by:-

$e ::= v \mid \mathbf{true} \mid \mathbf{false} \mid \overline{e_1} \mid e_1 + e_2 \mid \ \mathsf{case}\ e_0\ \mathsf{of}\ \mathbf{true} : e_1, \mathbf{false} : e_2$

Having extended the syntax, we also extend the semantics. The basic Syntactic and Operational classes remain otherwise unaltered and the semantics of the case statement is given by the rules in Figure 7.2.

The next step is to define the syntax for the Logic+Delay language itself. The extended language uses the expression syntax, Expr, as before, adding in three new syntax classes: BindExpr, Binding and Circuit.

Language Classes

Bind Expression $(rhs \in BindExpr)$

$rhs ::= e \mid \Delta_d(e) \qquad$ (where $d \in \mathbb{N}$)

Binding $(bd \in Bind)$

$bd ::= (v = rhs)$

Circuit $(cir \in Circ)$

$cir ::= \{bd_1, \cdots, bd_n\}$

Thus a BindExpr is either an Expr or a delayed Expr with the (stable) delay given by the positive integer subscript, d. A Binding sets a port name to the value of a BindExpr and a Circuit consists of a set of Binding items. No other conditions are imposed on the port names bound within a Circuit such as "define before use". For example, this means that we can specify circuits containing loops, so providing systems with fan-out and feedback signals.

7.4.1 Semantics of LD

The semantics of LD involves several kinds of execution, each introduced in terms of appropriate transition relations. Before doing this, we need to introduce some further Operational classes:

$$\begin{array}{rcll} \mathbb{B}_{\triangle} & = & \mathbb{B} \cup \{?\} & \text{(ranged over by } bv) \\ State & = & Var \xrightarrow{m} \mathbb{B}_{\triangle} & \text{(ranged over by } s) \\ StHist & = & State^{+} & \text{(ranged over by } h) \end{array}$$

The value set $\mathbb{B}_{\triangle}$ is the two element type, $\mathbb{B}$, together with an additional "don't know" element, denoted by '?'. It is used to define those cases where the choice of a particular value from the two element type is not determined. Such a quantity cannot be tested for and may only be returned as a result. The use of '?' in this class follows the format of the "don't know" value provided in the ELLA language.

The set of state mappings, *State*, gives an instantaneous snapshot of the stable values associated with the ports that may occur in an expression. The *state history*, *StHist*, records the history of all states for a particular execution and is adequately modelled as a (non empty) sequence of state mappings.

Using the above, we can introduce the first of our transition relations for LD, which defines how a BindExpr is executed to produce a value:

$$(\_ \vdash_{\_} \_ \Rightarrow \_) \quad \subseteq \quad (StHist \times \mathbb{N} \times BindExpr \times \mathbb{B}_{\triangle})$$

where ($h \vdash_t e \Rightarrow b$) means that, at time t, expression e is executed to produce value b, based upon state history h. This relation is defined by the rules given in Figure 7.3.

$$\boxed{\text{BE:1}}\ \frac{h_t \vdash e \Rightarrow b}{h \vdash_t e \Rightarrow b}$$

$$\boxed{\text{BE:2}}\ \frac{d \leq t, \quad \bigwedge_{i \in \{t\text{-}d..t\}} h \vdash_i e \Rightarrow b}{h \vdash_t \Delta_d(e) \Rightarrow b}$$

$$\boxed{\text{BE:3}}\ \frac{h \not\vdash_t rhs \Rightarrow 0\,, \qquad h \not\vdash_t rhs \Rightarrow 1}{h \vdash_t rhs \Rightarrow ?}$$

Figure 7.3: Rules for executing BindExpr

Rule BE:1 show how a pure, undelayed expression e is evaluated, at time t in the context of a state history h.

The type of delay formalised by rule BE:2 is a variant of an ELLA 'ambiguity' delay. This is a form of *stable* delay whose output is a delayed version of the input, provided the input remains stable over the delay period. If it fails to remain stable then the output is '?'. Note that if the delay period $d = 1$ then it is easily seen that $\Delta_1(e)$ acts like a simple unit time transport delay. The full range of ELLA delays have already been investigated and their semantics defined axiomatically in terms of predicate calculus[HW89].

Rule BE:3 is a *meta* rule stating that if it is not possible to produce either Boolean value by evaluation, then the value '?' is returned.

We also require some way of elaborating sets of bindings and so we introduce the following transition relation:

$$(\_ \xrightarrow{\_} \_) \quad \subseteq \quad (StHist \times Circuit \times StHist)$$

where ($h \xrightarrow{cir} h'$) states that history h is transformed to h' according to the bindings in the set *cir*.

Finally, we need a transition relation for extending a state history by the next state, on the application of the next inputs:

$$(\_ \underset{\_}{\overset{\_}{\rightsquigarrow}} \_) \quad \subseteq \quad (StHist \times Circuit \times Inputs \times StHist)$$

where ($h \underset{I}{\overset{cir}{\rightsquigarrow}} h'$) shows how, for the next inputs I, the history h maps to h', under the given circuit cir. Assumptions concerning the treatment of the input are discussed below.

The inference rules for defining the above relations are given in Figure 7.4. The definitions of auxiliary functions such as the *history update* function,

$$(\_\dagger\dagger\_) : ((A \xrightarrow{\sim} B)^* \times (A \xrightarrow{\sim} B)) \longrightarrow (A \xrightarrow{\sim} B)^*$$

are given in the Appendix.

$$\boxed{\text{LD:1}} \quad \frac{\bigwedge_{i \in \{1..k\}} h \vdash_{\mathbf{len}\, h} e_i \Rightarrow b_i}{h \xrightarrow{\{v_i = e_i \mid i \in 1..k\}} h \dagger\dagger \{v_i \mapsto b_i \mid i \in 1..k\}}$$

$$\boxed{\text{LD:2}} \quad \frac{\begin{array}{l} 1.\ h' = (h \frown last(h)) \dagger\dagger I \\ 2.\ h' \xrightarrow{cir} \cdots \xrightarrow{cir} h'' \\ 3.\ h'' \xrightarrow{cir} h'' \end{array}}{h \underset{I}{\overset{cir}{\rightsquigarrow}} h''}$$

Figure 7.4: Rules for elaborating bindings

7.4.2 Explanation

We briefly describe how the rules given in Figure 7.4 are designed to work. We note that the system evolution over time is defined by the values assumed by the identifiers standing for signal ports, as mentioned by the circuit or by their use as input signals. Consequently, it is this state history that characterises the behaviour of the specified device. The other determinants of step-by-step behaviour are clearly the circuit itself and the next vector of input signals. The transition relation ($h \underset{I}{\overset{cir}{\rightsquigarrow}} h'$) describes this step-by-step evolution of the device's behaviour.

So, suppose that we have a state history, h, and we consider just one of the bindings, $id = e$, from a given circuit *cir*. By using the relation ($h \underset{e}{\overset{cir}{\rightsquigarrow}} b$), we can work out the new value, b, for the port *id* in the succeeding environment, h'. The transition relation ($h \xrightarrow{cir} h'$) is defined by rule LD:1 to compute, simultaneously, all of the updates corresponding to one of these evaluation stages. Note that, by using the ($\_ \dagger\dagger \_$) operator, rule LD:1 only changes the *last* element of any signal history, so ensuring that the past history remains unaltered, an intuitively obvious requirement!

The above shows how to perform one computation of new values for ports from the previous values. However, the values of the ports could also have been used elsewhere in the circuit, during the present time interval. Accordingly, it is necessary to *push* or propagate these values through again to see if there are any further changes in state. If the system is stable, this *ripple-through* effect caused by propagating new values terminates to produce a state history, h'', which is left unchanged by the updating process.

Due to this ripple-through effect, we need to compute the *fixpoint* of the transition relation, ($\_ \xrightarrow{cir} \_$), reachable by iteration from the original state history, h, but updated by the next input mapping. This propagation of values is achieved for a stable system by use of rule LD:2. If the system is not stable, then rule LD:2 cannot be applied and no result is obtained.

By the above informal argument, we claim that the system could only reach a stable state precisely when such a fixpoint exists. In addition, note that both our semantics do *not* require any of the usual fixpoint machinery that accompanies the use of Denotational Semantics to define programming languages - we are only dealing with finite relations over discrete valued structures.

Turning now to the question of the finiteness of systems specified within LD, we can see that system behaviour is determined by the state history. Since these can be of arbitrary length, each system has apparently an infinite state space. However, we informally show that this is not the case by arguing that the state histories could always be truncated suitably so that they only contain a finitely bounded amount of information affecting the behaviour of the system.

We first of all note that the set of possible state mappings is finite as each state is made up of a finite mapping from ports to a finite range of values. Also, each binding of value to a port depends upon previous values, possibly after some finite delay. Therefore, one may compute the maximum possible delay over all the bindings in a given circuit. This means that we need only use a bounded suffix of the state history whose length is at most $(1 + MaxDelay)$. Therefore the unbounded set of state histories may be replaced by a bounded set of *effective* state histories, *EffStHist*. Moreover this suggests that the cardinality of *EffStHist* is bounded above by:

$$
\begin{aligned}
|\,EffStHist\,| &\leq (\text{No. of state mappings})^{(1+MaxDelay)} \\
&= (|\,Vals\,|^{\,Sigs})^{(1+MaxDelay)} \\
&= |\,Vals\,|^{\,(Sigs \cdot\, (1+MaxDelay))}
\end{aligned}
$$

where:

Vals = the set of possible values associated with the ports
Sigs = the total number of ports occurring in the circuit, cir
MaxDelay = the maximum possible delay over cir

It is relatively easy to give examples of systems whose state space meets this bound for arbitrary values of *Vals*, *Sigs* and *MaxDelay*. For example, consider a ring counter which can be controlled by some logical combination of delayed inputs. Examples of this kind suggest that the bound given above is tight.

However, what is perhaps interesting from the point of view of verification is that the bound need *not* be attained in practice and that actual systems may possess many *fewer* states than the above implies.

7.4.3 Loop Checking and Instability

It is shown above that systems defined using LD are essentially finite-state systems and so have finite latency (or memory). However, systems specified in LD can be susceptible to instability. In this section, we carefully distinguish between *internal instability*, due to the failure to find a fixpoint of value propagation within a single time interval, and the possible lack of *output constancy* for a particular system (see below). The first sort of behaviour is certainly an error and should never occur in a correct design. However, the second sort of behaviour may or may not be undesirable, depending upon the nature and specification of the system.

As these systems are essentially finite-state, this suggests that a decidable rule for internal instability could be formulated as already done above for the case of stable behaviour. One possible rule is given below:

$$\boxed{\text{LD:3}}\ \frac{\begin{array}{l} 1.\ h' = (h^\frown last(h))\dagger\dagger I \\ 2.\ h' \xrightarrow{cir} \cdots \xrightarrow{cir} h'' \\ 3.\ h'' \xrightarrow{cir} h''' \xrightarrow{cir} \cdots \xrightarrow{cir} h'' \\ 4.\ h'' \neq h''' \end{array}}{h \underset{I}{\overset{cir}{\rightsquigarrow}} (h^\frown(\text{UNDEF}\dagger\ I))}$$

Rule LD:3 works in much the same way as for rule LD:2, except that instead of detecting a fixpoint, it detects a loop of length strictly greater than 1. It does this by finding a state history h'' occurring more than once, whose successor h''' is different from h''.

The conclusion of rule LD:3 says that the resulting state history is set to the original history, h, appended to by the state UNDEF, mapping all (relevent) identifiers to '?' but then overwritten by the input mapping, I. The justification for this "all or nothing" approach is that, since at least one part of the state failed to stabilise, the fact that any other part (except the inputs) did succeed in stabilising is regarded as somewhat incidental, and so they may as well be treated as unstable.

The causes of any instability, could be determined by investigating the iterated behaviour of the relation ($\_ \xrightarrow{cir} \_$) and then observing in which state components the looping phenomena actually arise.

It is clear that the rules LD:2 and LD:3 are mutually exclusive since a fixpoint (if it exists) is simply a loop of length 1. We also note that the relation ($\_ \xrightarrow{cir} \_$) is functional, for a fixed binding set *cir*, since expression execution is functional and hence deterministic. Therefore, since the relation ($\_ \xrightarrow{cir} \_$) can only update state histories by using a finite number of state mappings, there can only be a finite number of iterations before a loop occurs, i.e. at most $|\,Vals\,|^{\,Sigs}$ iterations, although in practice this limit may be significantly smaller.

7.4.4 Output Constancy

The notion of *output constancy* tries to capture the property that, if the inputs were held constant *during all succeeding time intervals*, the outputs would not cyclically change, but would *eventually* stabilise to some constant values. Such a property would certainly be required of, say, a multiplier circuit or other circuits for data and signal processing.

Surprisingly, it is not a mandatory requirement that all systems should have the output constancy property. For instance, if the inputs had been *clocked*, so that all major transitions of the system and its sub-components were synchronised together, then this property would certainly not be required since each major transition involves a fresh input sample. As we have not completely fixed our model of input change and its relationship to other timing issues, we do not enforce output constancy within our semantics. However, for the remainder of this section, we consider its formalisation in terms of our semantic models.

It is quite a subtle constraint to formalise since it involves looking ahead to check that an eventuality property holds over the evolving system behaviour, on the assumption that no further change in input would occur. In principle, this appears to involve infinite lookahead and may potentially be an undecidable property to check. However, by recourse to our observations concerning finiteness, we argue that a finite bound to the lookahead can be determined in advance so that, after all, it is decidable.

Before giving its formal characterisation, we first consider why a designer might want output constancy to hold rather than another similar property. Suppose that we had merely required the property that if the input is held constant over successive time intervals, then there should be no change in output (i.e. it remain constant). This clearly rules out oscillatory behaviour — but it also rules out much more as well, such as the behaviour of a finite buffer of latency $k > 0$. This could take at most k steps to produce a constant output, depending on what the previous k inputs had been. More generally, any output with non-zero latency would be declared illegal according to such a simple rule.

To formalise this output stability property, we introduce yet another transition relation:

$$(\_ \xrightarrow[\_]{\_} \_) \subseteq (StHist \times Circuit \times Inputs \times StHist)$$

where $(h \xrightarrow[I]{cir} h')$ shows how, for the next input mapping I, the history h *stabily* maps to h', under the given binding set cir.

The rule OC:1 below defines $(\_ \xrightarrow[\_]{\_} \_)$ in the case where the behaviour is stable:

$$\boxed{\text{OC:1}}\quad \frac{\begin{array}{l} 1.\ \ h \overset{cir}{\underset{I}{\rightsquigarrow}} h' \\[1ex] 2.\ \ h' \overset{cir}{\underset{I}{\rightsquigarrow}} \cdots \overset{cir}{\underset{I}{\rightsquigarrow}} h'' \\[1ex] 3.\ \ \text{for all } h''' \bullet (h'' \overset{cir}{\underset{I}{\rightsquigarrow}} \cdots \overset{cir}{\underset{I}{\rightsquigarrow}} h''') \end{array}}{h \xrightarrow[I]{cir} h'}$$

Notice how the relation $(\_ \xrightarrow[\_]{\_} \_)$ is a sub-relation of $(\_ \overset{\_}{\underset{\_}{\rightsquigarrow}} \_)$ obtained by adding the lookahead conditions (2.) and (3.) as constraints. Essentially, condition (2.) finds a successor state history h'' for h' so that, by condition (3.), every successor, h''', to h'' has the same end states. (The universal quantifier for h''' is necessary since negative occurrences of free variables like those only occurring (positively) in the premisses are otherwise implicitly existentially quantified.)

As in our previous discussions of constancy, we can also give a rule that covers the corresponding unstable case as well.

$$\boxed{\text{OC:2}}\quad \frac{\begin{array}{l} 1.\ \ h \overset{cir}{\underset{I}{\rightsquigarrow}} h' \\[1ex] 2.\ \ \text{for all } h'' \bullet ((h' \overset{cir}{\underset{I}{\rightsquigarrow}} \cdots \overset{cir}{\underset{I}{\rightsquigarrow}} h'') \Rightarrow \ differ(cir, I, h'')\ ) \end{array}}{h \xrightarrow[I]{cir} (h \frown (\text{UNDEF} \dagger\ I))}$$

where UNDEF is the state that maps all identifiers to '?'

The essence of this rule is that the second premiss corresponds to the *failure* of the second and third constraints for rule OC:1. The subsidiary relation:

$$(\ differ(\_, \_, \_)\ ) \subseteq (Circuit \times Inputs \times StHist)$$

is defined by the following rule:

$$\boxed{\text{U:1}}\ \frac{(h \overset{cir}{\underset{I}{\rightsquigarrow}} \cdots \overset{cir}{\underset{I}{\rightsquigarrow}} h'), \qquad last(h) \neq last(h')}{differ(cir, I, h)}$$

This predicate says that, eventually, there exists a successor state history, h', to h which, for constant input I, has a different end state.

7.5 LD Example Calculation

In this section we demonstrate how the LD rules operate by examining a RS flip flop. We take the following definition as our example:

$$\{c = \overline{\mathrm{d} + \mathrm{a}},\ d = \Delta_1(\overline{\mathrm{c} + \mathrm{b}}),\ e = \Delta_1(\overline{\mathrm{a}} + c)\}$$

and for convenience, we shall refer to this circuit by the name ff_{LD}. Notice that we have included a delay element within the feedback path - this accords with intended practice in the ELLA language from which LD is abstracted.

Any state mapping for ff_{LD} must contain value assignments for at least the portnames a, b, c, d and e. As before, ports a and b correspond to inputs and ports c, d and e correspond to outputs.

We shall now use the semantic rules for LD to progressively calculate the signal history as it evolves under the same sequence of inputs used earlier. To do this, we shall start with the following state as the first element in the signal history:

$$h_0 = \begin{array}{|c|c|} a & 0 \\ b & 0 \\ c & ? \\ d & ? \\ e & 1 \\ \hline & 0 \\ \hline \end{array}$$

Suppose that the initial inputs are:

$$I_1 = \begin{array}{|c|c|} \hline a & 1 \\ b & 0 \\ \hline \end{array}$$

To calculate h_1, the immediate extension of h_0, we shall need to deduce that the relation $(\ h_0 \overset{\mathsf{ff}_{LD}}{\underset{I_1}{\rightsquigarrow}}\ h_1\ )$ holds. So, to derive a suitable value for h_1, we shall have to apply rule LD:2 which, in turn, involves satisfying its three premisses.

To satisfy the first premiss of rule LD:2, we need to calculate an initial approximation to the next state in the history:

$$h_0{}' = \begin{array}{|c|cc|} a & 0 & 1 \\ b & 0 & 0 \\ c & ? & ? \\ d & ? & ? \\ e & 1 & 1 \\ \hline & 0 & 1 \\ \hline \end{array} = (h_0 {}^\frown last(h_0)) \dagger\dagger I_1$$

The second premiss of rule LD:2 then says to calculate the relation ($\_ \xrightarrow{ff_{LD}} \_$), starting from $h_0{}'$, until a history $h_0{}''$ is reached which satisfies the third premiss of rule LD:2; this history is then set equal to h_1.

To do the successive calculation of the relation ($\_ \xrightarrow{ff_{LD}} \_$) implied by the second premiss, we need to apply rule LD:1 repeatedly. This calculation will be recorded by simply noting the intermediate histories, ih_0, ih_1, etc. reached via successive uses of ($\_ \xrightarrow{ff_{LD}} \_$). As before, we shall suppress many of the details concerning the evaluation of expressions and bind expressions, and concentrate on the evolution of the circuit's behaviour.

So, first setting ih_0 to $h_0{}'$, we proceed by using rule LD:1 to show that
($ih_0 \xrightarrow{ff_{LD}} ih_1$) holds where:

$$ih_1 = \begin{array}{|c|cc|} a & 0 & 1 \\ b & 0 & 0 \\ c & ? & 0 \\ d & ? & ? \\ e & 1 & 1 \\ \hline & 0 & 1 \\ \hline \end{array}$$

Note that the new value for port c is always calculated in the *current* state of each intermediate state history, whereas the new value for port d is calculated using values from the immediately *previous* state in each intermediate history. Since previous states cannot change during computation of intermediate values, the value of delayed ports like d and e will not change as a result of computing intermediate histories using rule LD:1.

Because our 'or' operator, $(\_ + \_)$, is *non-strict* with respect to '?' (see figure 7.1), we are able to find a defined value for c. Specifically, this is because the value of $(? + 1)$ is equal to 1 according to rule E:4b. In addition, we can see that, using rule BE:3, the value of the BindExpr $(? + 0)$ is equal to ?, since none of the bind expression rules could reduce it to either 0 or 1. This implies that the result for port d will be '?', even though its value is delayed by one.

Now, having obtained ih_1, we repeat the calculation to produce:

$$ih_2 = \begin{array}{|c|cc|} a & 0 & 1 \\ b & 0 & 0 \\ c & ? & 0 \\ d & ? & ? \\ e & 1 & 1 \\ \hline & 0 & 1 \\ \hline \end{array}$$

We find that $ih_1 = ih_2$, and so ih_1 satisfies the third premiss and is set equal to $h_0{}'' = h_1$:

$$h_1 = \begin{array}{|c|cc|} a & 0 & 1 \\ b & 0 & 0 \\ c & ? & 0 \\ d & ? & ? \\ e & 1 & 1 \\ \hline & 0 & 1 \\ \hline \end{array}$$

So, by using rule LD:2 with the above data, we have shown that
($h_0 \overset{ff_{LD}}{\underset{I_1}{\rightsquigarrow}} h_1$) holds.

To continue, suppose that the next inputs to be supplied are:

$$I_2 = \begin{array}{|c|c|}\hline a & 0 \\ b & 0 \\ \hline\end{array}$$

As before, we need to apply rule LD:2 to calculate the successor history, h_2. So proceeding as before, we initiate the calculation with:

$$h_1{}' = \begin{array}{|c|ccc|}\hline a & 0 & 1 & 0 \\ b & 0 & 0 & 0 \\ c & ? & 0 & 0 \\ d & ? & ? & ? \\ e & 1 & 1 & 1 \\ \hline & 0 & 1 & 2 \\ \hline\end{array}$$

as the initial approximation for h_2. As before, we need to successively compute the relation ($\_ \xrightarrow{\text{ff}_{LD}} \_$) in much the same way as before, but this time starting from $h_1{}'$. After an application of rule LD:1, we obtain ih_4 as result:

$$ih_4 = \begin{array}{|c|ccc|}\hline a & 0 & 1 & 0 \\ b & 0 & 0 & 0 \\ c & ? & 0 & ? \\ d & ? & ? & 1 \\ e & 1 & 1 & 0 \\ \hline & 0 & 1 & 2 \\ \hline\end{array}.$$

Note that port c becomes '?' because it uses the current value of port d (at time 2) which is '?'. The next value for port d uses the values of ports a and c from the previous state (at time 1) and these are both stable. As ih_4 is different from $h_1{}'$, we recompute ($\_ \xrightarrow{\text{ff}_{LD}} \_$) starting from ih_4 to obtain ih_5 by an application of rule LD:1:

$$ih_5 = \begin{array}{|c|ccc|}\hline a & 0 & 1 & 0 \\ b & 0 & 0 & 0 \\ c & ? & 0 & 0 \\ d & ? & ? & 1 \\ e & 1 & 1 & 0 \\ \hline & 0 & 1 & 2 \\ \hline\end{array}.$$

As ih_5 is not equal to ih_4, we repeat the above calculation to obtain $ih_6 = ih_5$. Putting $ih_5 = h_1{}''$ we see that the third premise for the application of rule LD:2 is satisfied showing that ($h_1{}'' \xrightarrow{\text{ff}_{LD}} h_2$) holds, where:

$$h_2 = \begin{array}{|c|ccc|}\hline a & 0 & 1 & 0 \\ b & 0 & 0 & 0 \\ c & ? & 0 & 0 \\ d & ? & ? & 1 \\ e & 1 & 1 & 0 \\ \hline & 0 & 1 & 2 \\ \hline\end{array}.$$

As before, this process can be continued to build up the state history, as driven by the input sequence. For each epoch, we will need to successfully apply rule LD:2 and satisfy its three premisses. To do so, rule LD:1 will be used to successively compute intermediate state histories until a fix-point solution is reached, assuming that one exists.

Proceeding with these calculations in applying the following input sequence:

$I_{1..9} =$

a	1	0	0	0	1	1	1	0	1
b	0	0	0	1	1	1	0	0	0
	1	2	3	4	5	6	7	8	9

the following history results:

$h_9 =$

a	0	1	0	0	0	1	1	1	0	1
b	0	0	0	0	1	1	1	0	0	0
c	?	0	0	0	0	0	0	0	0	0
d	?	?	1	1	1	0	0	0	1	1
e	1	1	0	1	1	1	0	0	0	1
	0	1	2	3	4	5	6	7	8	9

by successive applications of rules LD:2 and LD:1.

7.6 Extending LD towards *BKE*

As alluded to earlier in section 7.2, we shall now sketch how this semantics of LD can be extended towards a history semantics for *BKE*. As mentioned earlier, the main difference between *BKE* and the LD language is that *BKE* allows the definition and use of functions and has an extended form of the `case` notation. In keeping with the policy of minimal change, we shall just indicate any additional rules required, together with any other changes to the existing system.

7.6.1 Extending Boolean expressions

We need to extend the Boolean expressions to include the form of `case` required for *BKE* as well as the application of defined functions. Also, it is more convenient to put the delay operator into the expression class, Expr and do away with BindExpr. The new expression syntax becomes:

Expressions, $(e \in \mathit{Expr})$ given by:-

$$e ::= v \mid \mathbf{true} \mid \mathbf{false} \mid \Delta_d(e_1) \mid \mathit{fnm}(e_1, \cdots, e_n) \mid \ \texttt{case}\ \mathit{row}\ \texttt{of}\ \mathit{citm}_1, \ldots, \mathit{citm}_n\ \texttt{end}$$

where $\mathit{fnm} \in \mathit{Id}$ and $d \in \mathbb{N}$.

Row expressions $(\mathit{row} \in \mathit{Row})$ given by:-

$$\mathit{row} ::= (e_1, \cdots, e_{n+1})$$

Case items $(\mathit{citm} \in \mathit{CaseItem})$ given by:-

$$\mathit{citm} ::= (\mathit{cv}_1, \cdots, \mathit{cv}_{k+1}) : e$$

Choice value items $(\mathit{cv} \in \mathit{ChVal})$ given by:-

$$\mathit{cv} ::= \mathbf{true} \mid \mathbf{false} \mid \ \mathsf{X}$$

Binding ($bd \in Bind$) given by:-

$$bd ::= (v = e)$$

Circuit ($cir \in Circ$)

$$cir ::= \{bd_1, \cdots, bd_n\}$$

In addition, we have to introduce the notion of function definition as follows:-

Function declaration ($\mathit{fn\text{-}d} \in FunDecl$) given by:-

$$\mathit{fn\text{-}d} ::= \textsf{fun}\ \mathit{fnm}(v_1, \cdots, v_n)e$$

Closure ($p \in Closure$) given by:-

$$cls ::= \mathit{fn\text{-}d}_1, \cdots, \mathit{fn\text{-}d}_n, cir$$

7.6.2 Semantic classes

To give the semantics of this extension, we need to add in function declaration environments to allow defined functions to be used in expressions.

$$\begin{array}{lcll} SemEnv & = & FnEnv \times StHist & \\ FnEnv & = & Id \xrightarrow{m} FnClosure & \text{(ranged over by } \mathcal{F}) \\ FnClosure & = & Var^{+} \times Expr & \text{(ranged over by } \mathit{fn\text{-}cls}) \end{array}$$

The function closures, FnClosure, is a representation of a declared function consisting of the formal parameter list and an expression denoting the body.

7.6.3 Transition relations

Each of the transition relations introduced earlier needs to be redefined to take the above changes into account. Many of the details are straightforward and can be easily extrapolated from the earlier work. In this section, we shall therefore present the signatures of the transition relations required and the additional rules required to handle function definitions.

$$\begin{array}{rcl} (\,(\_,\_) \vdash \_\ \_ \Rightarrow \_\,) & \subseteq & (SemEnv \times \mathbb{N} \times Expr \times \mathbb{B}_{\triangle}) \\ (\,(\_,\_) \vdash \_\ \_ \Rightarrow_{\text{row}} \_\,) & \subseteq & (SemEnv \times \mathbb{N} \times Expr^{+} \times \mathbb{B}_{\triangle}^{+}) \\ (\,\_ \vdash \_ \xrightarrow{\ \_\ } \_\,) & \subseteq & (FnClosure \times StHist \times Circuit \times StHist) \\ (\,(\_,\_) \overset{\_}{\rightsquigarrow} \_\,) & \subseteq & (SemEnv \times Circuit \times Inputs \times StHist) \\ (\,\_ =\!\boxed{\text{fun decl}}\!\Rightarrow \_\,) & \subseteq & (Closure \times FnEnv) \end{array}$$

7.6.3.1 Function declaration and application

To handle the declaration of functions, we need the following rule:

$$\boxed{\text{FnDcl:1}}\;\frac{}{[\,] \overset{\boxed{\text{fun decl}}}{\Longrightarrow} \emptyset}$$

$$\boxed{\text{FnDcl:2}}\;\frac{\mathit{fn\text{-}dcls} \overset{\boxed{\text{fun decl}}}{\Longrightarrow} \mathcal{F}}{[\,\mathsf{fun}\, \mathit{fnm}(v_1,\cdots,v_n)e] ^\frown \mathit{fn\text{-}dcls} \overset{\boxed{\text{fun decl}}}{\Longrightarrow} \{\mathit{fnm} \mapsto ((v_1,\cdots,v_n),e)\} \cup \mathcal{F}}$$

In addition, some well-formedness rules concerning 'define before use' can be given to forbid definition by recursion — this is enforced within ELLA style languages. By doing so, this ensures that the evaluation process involving function application will always terminate.

In order to apply a function, we add the following rule:

$$\boxed{\text{fun apply}}\;\frac{\begin{array}{l}1.\;\; \mathcal{F}(\mathit{fnm}) = ((v_1,\cdots,v_k),e_v) \\ 2.\;\; (\mathcal{F},h) \vdash_t e_v[e_1/v_1,\cdots,e_k/v_k] \Rightarrow v\end{array}}{(\mathcal{F},h) \vdash_t \mathit{fnm}(e_1,\cdots,e_k) \Rightarrow v}$$

where the notation $e[e_1/v_1,\cdots,e_k/v_k]$ stands for simultaneous substitution of e_i for V_i throughout the expression e. This rule captures a 'copy rule' approach, where the actual parameter expressions are substituted directly for the formal parameters. It turns out this is *necessary* since formal parameters could occur inside delay expressions occurring in the body e — so precluding the possibility of evaluating the actual parameters independently from this context in the body. The termination of the function evaluation process is guaranteed since definition by recursion is forbidden (see earlier remarks).

7.6.3.2 Extended Cases Statement

As there are no primitive function constants, it is necessary to introduce a comprehensive case construct to allow such functions to be defined. The rules we need to add are as follows:

$$\boxed{\text{case}}\;\frac{\begin{array}{l}1.\;\; (\mathcal{F},h) \vdash_t \mathit{row} \Rightarrow_{\mathrm{row}} \vec{v} \\ 2.\;\; (\mathcal{F},h) \vdash_t \vec{v}\ \mathsf{matches}\ \mathit{c\text{-}itm}_i \Rightarrow v\end{array}}{(\mathcal{F},h) \vdash_t\ \mathsf{case}\ \mathit{row}\ \mathsf{of}\ \mathit{c\text{-}itm}_1,\cdots,\mathit{c\text{-}itm}_n\ \mathsf{end} \Rightarrow v}$$

$$\boxed{\text{case matching}}\;\frac{\begin{array}{l}1.\;\; \bigwedge_{i\in\{1..k\}} (v_i = cv_i) \vee (cv_i = \mathsf{X}) \\ 2.\;\; (\mathcal{F},h) \vdash_t e \Rightarrow v\end{array}}{(\mathcal{F},h) \vdash_t (v_1,\cdots,v_k)\ \mathsf{matches}\ ((cv_1,\cdots,cv_k),e) \Rightarrow v}$$

7.6.4 Towards a history semantics for the let construct

The above language does not *yet* have the let construct as required for *BKE*. The syntax of the let construct is as follows:

let *circ* in *expr*

The new technical complication is that the inner Circuit *circ* will require a signal history environment to interpret it due to the possible occurrence of delayed sub-expressions. Essentially, we will require fully general 'nested' fixpoints to arbitrary depth, since each element of the Circuit is a binding of the form:

$var = expr$

where the RHS *expr* may contain further occurrences of the let construct.

One possible (but non-compositional) approach is to completely flatten the nesting structure and then compute the result by a single fixpoint. This has a number of awkward, but not insurmountable, technical difficulties associated with it. Firstly, the unnesting of fixpoints is a subtle and complex task and requires extending the name structure in the signal history to include all the local names after they have been uniquely identified, perhaps by some pathname. Secondly, the introduction of this global structure will significantly complicate the comparison of the resulting history semantics with the (compositional) semantics provided in terms of IO automata. One approach to effecting this comparison might be via the use of an observation function defined on the history semantics to hide both the effect of unbounded history and the additional local variables.

Such matters will need further study and consideration at a later stage of the project.

Chapter 8

Comparison of Automaton and History Semantics

This chapter is intended to contain a comparison between the two semantics for *BKE* given in earlier chapters. Clearly, such a comparison cannot be made until both semantics have been completed for *BKE*.

Chapter 9

Extending Boolean Kernel ELLA to Kernel ELLA

This chapter is intended to show how *BKE* will be extended into Kernel ELLA. Broadly, the requirement is to generalise the value space to contain general structured values, rather than merely Booleans. As such, it indicates the implications on the IOA semantics of such an extension.

As such, this will form part of further work during the project.

Part IV

Verification

Chapter 10

ELLA to ELLA Verification

This chapter begins with a summary of the approach to ELLA to ELLA verification. An algorithm for the trace equivalence of ELLA systems is discussed. The main part of this chapter contains two case studies - the SR latch and the twisted ring counter.

10.1 The Approach

In chapter 2 on Verification Requirements, it was pointed out that most users questioned required support for some form of equivalence checking between ELLA designs. Current practice provides for comparison of results obtained from exhaustive simulation of the two designs. This is an extremely time-consuming process and often is not complete. However, the particular approach does lend itself to comparing non-ELLA and ELLA designs — say, gate-level ELLA with a high-level design in, for example, C. In the case of comparing two ELLA designs, formal support is readily available. The approach is to compare the two designs via their respective formal semantic descriptions. Of the two semantic styles presented in part II, the history based semantics is not so useful in this respect[1]. However, in the case of the automaton based semantics, the task of comparison becomes one of testing equivalence or containment of the automata representing the designs. By that we mean check that the languages accepted / generated by the automata are the same. Now as there are "standard" automata-theoretic algorithms for performing complementation, intersection and testing emptiness of automata, testing for equivalence is somewhat simplified. We outline the algorithms in the next section.

10.2 Algorithm for Trace Equivalence and Containment

In this section, we shall describe functional algorithms for trace equivalence and containment of IO automata. We shall base these upon early work by E.F. Moore [Moo56] to obtain efficient algorithms in the case where there is no eventuality constraints upon the *infinite* traces that are accepted (i.e. any infinite looping run is accepted). As such restrictions do not naturally arise within ELLA itself, we can use these algorithms to check for trace equivalence and containment between ELLA descriptions.

To maintain clarity and a direct mathematical interpretation, we shall write our algorithms in functional terms, and briefly make some observations concerning how a more concrete implementation might be obtained.

[1] A theorem prover for the meta logical language used for the semantics would be required.

10.2.1 Trace containment and equivalence

Essentially, the notions of trace containment and equivalence are defined by means of the following definitions:

$$Env = Name \xrightarrow{m} Val$$

$$ILabel = Env$$

$$OLabel = Env$$

$$IOLabel = (ILabel \times OLabel)$$

$$Runs : IOA \to \mathbb{P}(Trans^*)$$

$$\begin{array}{l} Runs(mk\text{-}IOA(sts, im, om, trns, st)) \quad \triangleq \\ \quad \{rn \in Trans^* \\ \quad \mid start(rn[1]) \in st \wedge (\forall i \in \textbf{inds}\, rn \cdot i < \textbf{len}\, rn \;\Rightarrow\; finish(rn[i]) = start(rn[i+1]))\} \end{array}$$

$$trace : Trans^* \to IOLabel^*$$

$$\begin{array}{l} trace(rn) \quad \triangleq \\ \quad \textbf{cases}\ rn\ \textbf{of} \\ \quad \textbf{nil} \qquad\quad \to \textbf{nil} \\ \quad [tn] \frown rn' \to [(ivals(tn), ovals(tn))] \frown trace(rn') \\ \quad \textbf{end} \end{array}$$

$$IOLang : IOA \to \mathbb{P}(IOLabel^*)$$

$$IOLang(ioa) \quad \triangleq \quad \{trace(rn) \mid rn \in Runs(ioa)\}$$

The above definitions define what a legal run of an IO automaton is, from which we can extract the IO language expressed in terms of traces. This allows us to define the notions of trace containment and equivalence between IO automata:

$$is\text{-}trace\text{-}contained : IOA^2 \to \mathbb{B}$$

$$is\text{-}trace\text{-}contained(ioa_1, ioa_2) \quad \triangleq \quad IOLang(ioa_1) \subseteq IOLang(ioa_2)$$

$$is\text{-}trace\text{-}equivalent : IOA^2 \to \mathbb{B}$$

$$is\text{-}trace\text{-}equivalent(ioa_1, ioa_2) \quad \triangleq \quad IOLang(ioa_1) = IOLang(ioa_2)$$

10.2.2 Implementation relation

We may also define another useful comparison between IO automata - that is, when one automaton *is implemented by* another automaton. To do so, we need to define the notion of the IO language accepted by an automaton relative to some given input language:

$$ILang : IOA \rightarrow \mathbb{P}(ILabel^*)$$

$$ILang(ioa) \quad \triangleq \quad \{fst(unzip(tr)) \mid tr \in IOLang(ioa)\}$$

$$IOLangI : IOA \times \mathbb{P}(ILabel^*) \rightarrow \mathbb{P}(IOLabel^*)$$

$$IOLangI(ioa, in\text{-}lang) \quad \triangleq \quad \{tr \in IOLang(ioa) \mid fst(unzip(tr)) \in in\text{-}lang\}$$

where *fst* is the projection on the first argument and *unzip* simply maps a sequence of paired elements into a pair of sequences.

The *implementation relation* may now be defined:

$$is\text{-}implemented\text{-}by : IOA^2 \rightarrow \mathbb{B}$$

$$is\text{-}implemented\text{-}by(ioa_1, ioa_2) \quad \triangleq \quad IOLang(ioa_1) = IOLangI(ioa_2, ILang(ioa_1))$$

In essence, this says that ioa_2 implements ioa_1 whenever ioa_2, restricted to the input language for ioa_1, could do precisely the same input/output actions as can ioa_1. Therefore, ioa_2 is permitted to have additional behaviours that are unconstrained by the automaton ioa_1.

We observe that if ioa_2 is *deterministic*, then the implementation relationship degenerates to trace containment:

$$ioa_2 \text{ is deterministic } \Rightarrow$$
$$is\text{-}implemented\text{-}by(ioa_1, ioa_2) \Leftrightarrow is\text{-}trace\text{-}contained(ioa_1, ioa_2)$$

This is easily seen by noting that, since ioa_2 is deterministic, there would be at most one input/output trace of ioa_2, for any input trace of ioa_1.

10.2.3 Functional description of the algorithm

The basic idea is to trace through each of the machines and to match corresponding states, starting from the initial states. Essentially, we have to generate a certain kind of matching relation showing how the states of one machine correspond to the states of the other. This algorithm is essentially due to E.F.Moore [Moo56], at least for the case of comparing deterministic automata. However, our contribution here has been to adapt this algorithm to the non-deterministic case. This has been done by, in effect, merging Moore's algorithm with the standard *subset construction* due to Rabin and Scott [RS59] for "determinising" non-deterministic automata. What we then obtain is an algorithm that does 'on the fly' determinisation of the automata, during the process of matching states.

The heart of this algorithm revolves around setting up a binary matching relation between sets of corresponding states. Such a matching relation will belong to the following type:

$$MatchReln = (State\text{-}\mathbf{set} \times State\text{-}\mathbf{set})\text{-}\mathbf{set}$$

This relation is defined so that matching pairs of sets share a common trace prefix and should also have corresponding transitions. The main functions for computing the relation are as follows

$$trans\text{-}maps : Trans\text{-}\mathbf{set} \times State\text{-}\mathbf{set} \rightarrow IOLabel\text{-}\mathbf{set}$$

$$trans\text{-}maps(ts, sts) \quad \triangleq \quad \{(ivals(tn), ovals(tn)) \in IOLabel \mid \exists tn \in ts \cdot from(tn) \in sts\}$$

$$extend\text{-}trans : (Trans\text{-}\mathbf{set} \times Trans\text{-}\mathbf{set}) \to (ILabel \times OLabel \times State\text{-}\mathbf{set} \times State\text{-}\mathbf{set}) \to MatchReln$$

$$\begin{array}{l} extend\text{-}trans(ts_1, ts_2)(im, om, ms_1, ms_2) \quad \triangleq \\ \quad \{(rs_1, rs_2) \in (State\text{-}\mathbf{set} \times State\text{-}\mathbf{set}) \\ \qquad \mid rs_1 \ = \ image\text{-}trans(im, om, ts_1, ms_1) \ \wedge \ rs_2 \ = \ image\text{-}trans(im, om, ts_2, ms_2)\} \end{array}$$

$$image\text{-}trans : ILabel \times OLabel \times Trans\text{-}\mathbf{set} \times State\text{-}\mathbf{set} \to State\text{-}\mathbf{set}$$

$$\begin{array}{l} image\text{-}trans(im, om, ts, ms) \quad \triangleq \\ \quad \{to(tn) \in State \mid \\ \qquad tn \in ts \ \wedge \ from(tn) \in ms \wedge \ ivals(tn) \ \subseteq im \wedge \ ovals(tn) \ \subseteq om \\ \quad \} \end{array}$$

$$ExpandMatchReln : (Trans\text{-}\mathbf{set} \times Trans\text{-}\mathbf{set}) \to MatchReln \to MatchReln$$

$$\begin{array}{l} ExpandMatchReln(ts_1, ts_2)(m\text{-}rel) \quad \triangleq \\ \quad m\text{-}rel \ \cup \\ \qquad \bigcup\{extend\text{-}trans(ts_1, ts_2)(im, om, ms_1, ms_2) \mid \\ \qquad\quad (im, om) \in (trans\text{-}maps(ts_1, ms_1) \ \cup \ trans\text{-}maps(ts_2, ms_2)) \wedge (ms_1, ms_2) \in m\text{-}rel \\ \qquad \} \end{array}$$

$$StepMatchReln : (Trans\text{-}\mathbf{set} \times Trans\text{-}\mathbf{set}) \to MatchReln \to MatchReln$$

$$\begin{array}{l} StepMatchReln(ts_1, ts_2)(m\text{-}rel) \quad \triangleq \\ \quad \mathbf{let}\ new\text{-}m\text{-}rel = ExpandMatchReln(ts_1, ts_2)(m\text{-}rel) \\ \quad \mathbf{in} \\ \qquad \mathbf{if}\ (new\text{-}m\text{-}rel \subseteq m\text{-}rel) \\ \qquad \mathbf{then}\ new\text{–}m\text{-}rel \\ \qquad \mathbf{else}\ StepMatchReln(ts_1, ts_2)(new\text{-}m\text{-}rel) \end{array}$$

$$ComputeMatchReln : IOA^2 \to MatchRel$$

$$\begin{array}{l} ComputeMatchReln(m_1, m_2) \quad \triangleq \\ \quad \mathbf{let}\ mk\text{-}IOA(\_, \_, \_, ts_1, sts_1) = m_1, \\ \qquad mk\text{-}IOA(\_, \_, \_, ts_2, sts_2) = m_2 \\ \quad \mathbf{in} \\ \qquad\quad StepMatchReln(ts_1, ts_2)(\{(sts_1, sts_2)\}) \end{array}$$

Note that the function *StepMatchReln* is sufficiently well-defined insofaras that $m\text{-}rel \subseteq new\text{-}m\text{-}rel$ by definition of *ExtendMatchReln*. Since the number of possible mappings fulfilling this condition is finite, we can conclude that *StepMatchReln* will eventually reach a fixpoint.

Moreover, note that the use of *trans-maps* will ensure that *im* and *om* are selected so that

$$(\emptyset, \emptyset) \notin expand\text{-}trans(ts_1, ts_2)(im, om, ms_1, ms_2)$$

and assuming that $(\emptyset, \emptyset) \notin m\text{-}rel$, we have that

$$(\emptyset, \emptyset) \notin ExpandMatchReln(ts_1, ts_2)(m\text{-}rel)$$

We may now use the above functions and predicates to define predicates corresponding to trace containment and equivalence:

$$Contained\text{-}By : IOA^2 \rightarrow \mathbb{B}$$

$$\begin{array}{l} Contained\text{-}By(m_1, m_2) \quad \triangleq \\ \quad \mathbf{let}\ m\text{-}rel = ComputeMatchReln(m_1, m_2)\ \mathbf{in} \\ \qquad \neg\ \exists\ (ms_1, ms_2) \in m\text{-}rel\ \cdot\ ms_1 \neq \emptyset\ \wedge\ ms_2 = \emptyset \end{array}$$

$$Equiv : IOA^2 \rightarrow \mathbb{B}$$

$$\begin{array}{l} Equiv(m_1, m_2) \quad \triangleq \\ \quad \mathbf{let}\ m\text{-}rel = ComputeMatchReln(m_1, m_2)\ \mathbf{in} \\ \qquad \forall\ (ms_1, ms_2) \in m\text{-}rel\ \cdot\ ms_1 \neq \emptyset\ \wedge\ ms_2 \neq \emptyset \end{array}$$

The correctness of the above may be seen by observing the following:

- By construction, sets of states are related to those corresponding sets of states in the other machine which share a common prefix trace.
- Hence, the matching relation only contains Ø in either its domain or range when there is some trace contained in one machine but not the other.
- Accordingly, the presence or absence of Ø in the matching relation can indicate equivalence, containment in one or other direction, or inequivalence.

10.2.3.1 Implementation aspects

Any efficiency gains from using a concrete implementation of this algorithm are likely to involve the following factors. First of all, the comparison and matching process can be arranged to follow the transition structure of one of the machines, so reducing the number of comparisons between machine states that could never be related on structural grounds. This is an immeadiate gain over constructions that use Cartesian product of machines and complementation. Secondly, we can also ensure that few comparisons are repeated by only expanding the match relation for those state set pairs that were *freshly* generated during the current iteration. This further implies that the test to check if a given set pair belongs to the matching relation will need to be efficient.

Thirdly, by keeping track of which machine fails to have transitions that the other machine has required, it is possible to say, **at the end of constructing the matching relation**, whether the machines are equivalent, one is contained in the other or if they are otherwise inequivalent. Therefore, this avoids duplication of matching effort when checking for equivalence – with other algorithms, such duplication is unavoidable. Overall, therefore, we expect to be able to check for conflicts and mismatches during the matching process. In addition, this will allow for the timely 'on the fly' generation of error reports or, alternatively, it allows for the checking process to halt as soon as the outcome is determined. Finally, we expect that there should be little additional overhead when either of the automata are already deterministic, as is to be expected when checking correct ELLA descriptions.

10.3 Example: SR Latch

In this section we give two *BKE* descriptions of the SR-latch. The first is a algorithmic description and the second a gate-level description in terms of Nor-gates. We describe the construction of the automaton for each of these descriptions and show that the semantic objects generated from the two descriptions are trace equivalent; in fact, they are isomorphic.

10.3.1 Algorithmic description

The SR-latch is described by the following *BKE* text

```
fn SR-FUN(r, s) =
 let q = case (r,s) of
          (1,0): 0,
          (0,1): 1,
          (1,1): 0
        else dq
        esac,
     dq = IDelay(q,0,1)
 in (q,¬ q).
```

To construct the automaton for this function, we first construct that for the case statement

```
case (r,s) of
  (1,0): 0,
  (0,1): 1,
  (1,1): 0
else dq
esac
```

Following the construction described in section 6.3, we construct the automaton for *chooser-ioa*. This is a single-state automaton, so all transitions are from state 0 to itself.[2]

In						Out
-1	-2	-3	-4	-5	-6	1
1	0	0				0
1	0	1				1
0	1		0			0
0	1		1			1
1	1			0		0
1	1			1		1
0	0				0	0
0	0				1	1

Inputs -1 and -2 will be 'connected', using the causal product, to the outputs of the selector expression ((r,s)), and the other inputs to the outputs of the choice actions.

The automaton for the selector is the single-state automaton

[2]In the following, blank entries for inputs denote that these inputs can take any value, and states will be omitted from the tables of single state automata.

In		Out	
r	s	1	2
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

The three automata for the choice actions are single-state automata with no inputs.

Out
1
0

Out
1
1

Out
1
0

and the else action has single state-automaton, which copies input to output.

In	Out
dq	1
0	0
1	1

Combining these using the causal product (after renaming) gives the single-state automaton

In			Out
r	s	dq	1
1	0		0
0	1		1
1	1		0
0	0	0	0
0	0	1	1

IDelay(q,0,1) has the two-state automaton (with initial state 0)

From	In	Out	To
	q	1	
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

We now construct the automaton for the let construction. The automaton for the returned value is the single-state automaton

In	Out	
q	1	2
0	0	1
1	1	0

The two automata whose causal product give the binding of the let clause are

In			Out
r	s	dq	q
1	0		0
0	1		1
1	1		0
0	0	0	0
0	0	1	1

From	In	Out	To
	q	dq	
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

The causal product of these two automata is

From	In		Out		To
	r	s	q	dq	
0	1	0	0	0	0
0	0	1	1	0	1
0	1	1	0	0	0
0	0	0	0	0	0
1	1	0	0	1	0
1	0	1	1	1	1
1	1	1	0	1	0
1	0	0	1	1	1

Since this automaton is the product of an automaton with states $\{0\}$ and another with states $\{0,1\}$, its states should have names $(0,0)$ and $(0,1)$. In order to simplify notation we have renamed them as 0 and 1; we adopt this approach consistently throughout this section. A simple renaming of inputs and outputs, and some merging of transitions, gives the following automaton for the function SR-ALG, shown in figure 10.1.

From	In		Out		To
	-1	-2	1	2	
0		0	0	1	0
0	0	1	1	0	1
0	1	1	0	1	0
1	1		0	1	0
1	0		1	0	1

10.3.2 Gate-level description

We now give a gate-level description of the SR-latch, whose circuit is shown in figure 10.2

```
fn INV(x)    = ¬ x,

fn NOR(x, y) = INV(x + y),

fn BDEL(y) = IDelay(y,0,1),
```

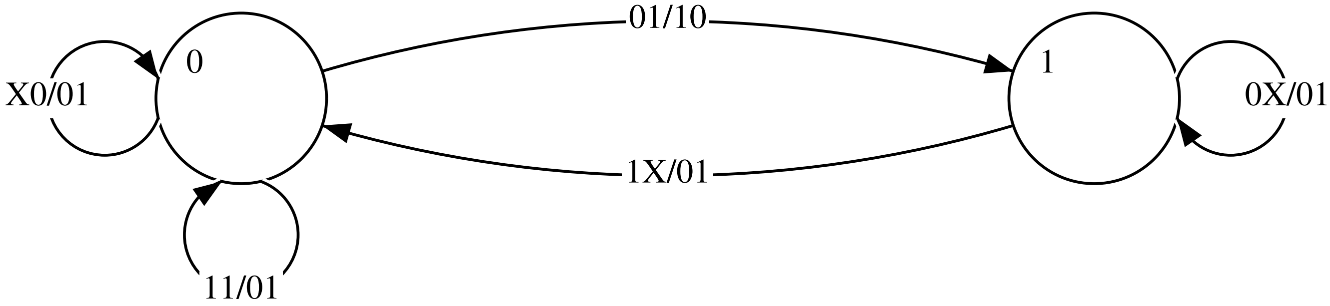

Figure 10.1: SR-latch

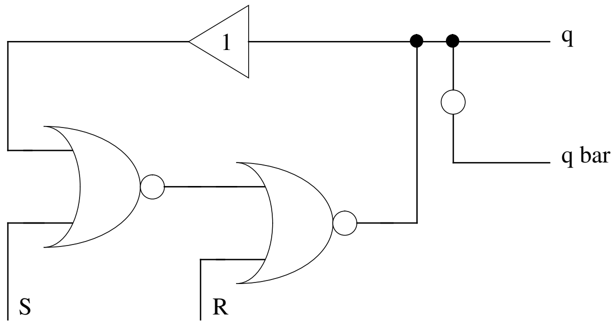

Figure 10.2: SR-latch

```
fn SR-GATE(reset, set) =
  let nor  = NOR(set,BDEL(q)),
      q    = NOR(reset,nor),
      qbar = INV(q)
  in (q,qbar).
```

The functions NOR and DEL have automata

NOR		
In		Out
-1	-2	1
0	0	1
0	1	0
1	0	0
1	1	0

BDEL			
From	In	Out	To
	-1	1	
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

so that NOR(set,BDEL(q)) is the product of the three automata

In		Out
-1	-2	1
0	0	1
0	1	0
1	0	0
1	1	0

In	Out
set	-1
0	0
1	1

From	In	Out	To
	q	-2	
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

which results in the table

From	In		Out	To
	set	q	1	
0	0	0	1	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	0
1	1	1	0	1

We now construct the automaton for the let construction. The automaton for the returned value is the single-state automaton

In		Out	
q	qbar	1	2
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

The three other automata required to give the binding of the let clause are

From	In		Out	To
	set	q	nor	
0	0	0	1	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	0
1	1	1	0	1

In		Out
reset	nor	q
0	0	1
0	1	0
1	0	0
1	1	0

In	Out
q	qbar
0	1
1	0

This gives the following automaton for the let expression

From	In		Out		To
	reset	set	q	qbar	
0	0	0	0	1	0
0	0	1	1	0	1
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	1	0	1	0

which finally gives the following for the function SR-GATE

From	In		Out		To
	-1	-2	1	2	
0		0	0	1	0
0	0	1	1	0	1
0	1	1	0	1	0
1	1		0	1	0
1	0		1	0	1

This is identical to the automaton derived for the function SR-ALG.

10.4 Example: Twisted Ring-Counter

Our second example to illustrate ELLA to ELLA verification is a three-bit twisted-ring counter[3]

Informally, a twisted ring counter, as in [BEGS89], cycles through a finite sequence of values. Here we restrict our attention to a 3-bit version of the counter. It has 6 valid values, corresponding to 6 valid states, and 2 invalid values (states). The normal counting cycle is through the 6 valid values, shown in Table 10.1, Invalid values can occur through device error, State 2: (0,1,0) and State 5: (1,0,1). However, should the counter enter an invalid state, the device will recover to a valid state within one, or two, steps. A reset is provided to force the counter to 0.

10.4.1 First Description of Twisted Ring-Counter

We now give "algorithmic" descriptions for this example, first in ELLA and then in *BKE*.

10.4.1.1 ELLA Description

[3]In [BEGS89] a three-bit twisted ring-counter is used as the basis of a case-study of formal specification simulation and verification. The purpose of this study was to compare a selection of specification and theorem-proving systems, namely Boyer-Moore, HOL, Temporal Logic and OBJ3. The ring-counter therefore serves as an ideal candidate to illustrate the language *BKE*, its formal IO automaton semantics and verification process, because the example design and sample verifications are small enough to describe completely within this paper, and results can be compared with those given in [BEGS89], which may appear in a more familiar form.

State Label	Outputs		
	a	b	c
0	0	0	0
1	0	0	1
3	0	1	1
7	1	1	1
6	1	1	0
4	1	0	0
0	0	0	0
.	.	.	.
.	.	.	.

Table 10.1: Sequence of Outputs

```
TYPE bool = NEW (t|f).

FN NEXT = (bool: reset, [3]bool: in) -> [3]bool:
CASE reset OF
     f : CASE in OF
              (f,f,f): (f,f,t),
              (f,f,t): (f,t,t),
              (f,t,t): (t,t,t),
              (t,t,t): (t,t,f),
              (t,t,f): (t,f,f),
              (t,f,f): (f,f,f),
              (f,t,f): (f,f,f),
              (t,f,t): (f,t,f)
         ELSE (f,f,f)
         ESAC,
     t : (f,f,f)
ELSE (f,f,f)
ESAC.

FN BOOL_DELAY = (bool) -> bool:
   DELAY(f,1).



FN DELAY_3 = ([3]bool: in) -> [3]bool:
 ( BOOL_DELAY(in[1],1),
   BOOL_DELAY(in[2],1),
   BOOL_DELAY(in[3],1)
 ).

FN COUNTER = (bool: reset) -> [3]bool:
```

```
BEGIN
  MAKE DELAY_3: tbd.
  JOIN NEXT (reset, tbd) -> tbd.
  OUTPUT tbd
END.
```

The counter is formed by applying a function, NEXT, which computes the next value to be output from the value previously output by the counter, given via the DELAY_3 function.

10.4.1.2 *BKE* Description

In *BKE* the high-level description is written using a recursive let.

```
fn NEXT(reset,p,q,r) =
case reset of
   0 : case (p,q,r) of
          (0,0,0): (0,0,1),
          (0,0,1): (0,1,1),
          (0,1,1): (1,1,1),
          (1,1,1): (1,1,0),
          (1,1,0): (1,0,0),
          (1,0,0): (0,0,0),
          (0,1,0): (0,0,0),
          (1,0,1): (0,1,0)
       else (0,0,0)
       end,
    1 : (0,0,0)
else (0,0,0)
end.

fn DELAY˙3(p,q,r) =
( Idelay(p,0,1), Idelay(q,0,1), Idelay(r,0,1) ).

fn COUNTER(reset) =
let (dp,dq,dr) = DELAY˙3(p,q,r),
   (p,q,r) = NEXT(reset,dp,dq,dr)
in  (p,q,r).
```

10.4.1.3 IO Automaton

Here we outline the construction of the IO automaton semantic object for the above *BKE* description.

The semantic object denoting the NEXT function is clearly a single-state automaton (it has no delays) with 16 transitions, each corresponding to the various choices of the case construct.

Next consider the semantic object that represents the rowed expression

```
( Idelay(p,0,1), Idelay(q,0,1), Idelay(r,0,1) ).
```

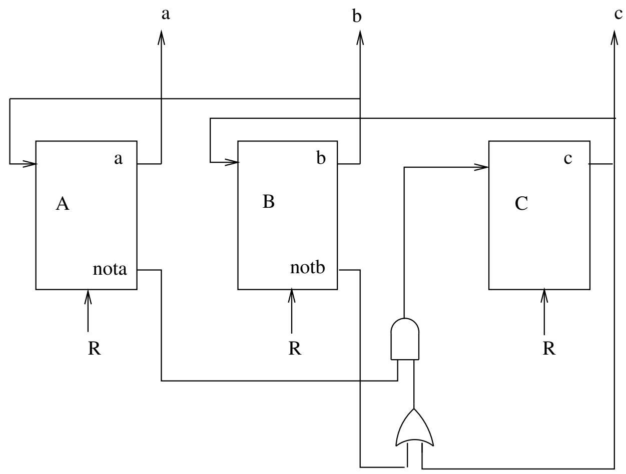

Figure 10.3: Twisted Ring-Counter

It is constructed by forming the causal product of the individual row-elements. Each element is in fact denoted by a two state automaton, similar to that given in figure 6.2 of section 6.2.1.2, except that the input to each is named by p, q, r and d respectively and the outputs are named 1, 2 and 3 respectively. The resulting product can have at most eight states. As the alphabets of each element are independent, and each state has a transition to every state (fully connected), the resulting automaton is also fully connected (the causal product degenerates to the usual cross product) and has 8 states and 64 transitions. This automaton is then transformed into an object denoting the function DELAY_3, essentially by renaming the inputs p, q and r by the internal names -1, -2 and -3.

The semantic object for the COUNTER is obtained by "solving" the mutually recursive equations of the let clause. This effectively renames the outputs of the eight state delay object (DELAY_3) to be the appropriate inputs of the next state automaton (NEXT) and then forming the causal product. This has the effect of restricting the transition of the 8 state automaton to be those allowed by the single-state NEXT automaton.

A picture of the resulting IO automaton is given in Figure 10.4

10.4.2 Gate-level Description of Twisted Ring-Counter

A "gate-level" design for the twisted ring-counter contains three latches with reset, one AND gate and one OR gate, as shown in Figure 10.3. There is a single input *reset* $\in \{0,1\}$ which resets the circuit to State 0 if it is raised. Note that it is coincidental and for convenience only that the states have labels corresponding to the integer values of the outputs on their departing arcs.

10.4.2.1 ELLA Description

An ELLA-text description is now given for the ring-counter, using a two-valued Boolean data type. The initial state is assumed here to be State 0. A "*MAKE...JOIN...OUTPUT*" description style has been adopted, and the description has been written to resemble as far as possible the *BKE* description which follows, so that equivalent parts in each text can be compared readily.

```
FN BOOL_DELAY = (bool) -> bool:
   # initial value f, delay 1 unit#
   DELAY(f,1).

FN OR = (bool: x y) -> bool:
CASE (x, y) OF
   (f, f): f,
   ((bool, t) | (t, bool)): t
ESAC.

FN NOT = (bool: x) -> bool:
CASE x OF
   f: t,
   t: f
ESAC.

FN AND = (bool: x y) -> bool:
   # defined in terms of NOT and OR: #
   NOT( NOT(x) OR NOT(y)).

FN LATCH = (bool: i r) -> bool:
# i is input, r is reset:   #
# out(n+1) = i(n) if r(n)=f #
# out(n+1) = f    if r(n)=t #

      BOOL_DELAY( i AND (NOT r)).

FN TWIST = (bool: reset) -> [3]bool:
# twisted ring-counter #
  BEGIN
     MAKE LATCH: a b c.
     JOIN (b, reset) -> a,
          (c, reset) -> b,
          (AND (NOT a, (NOT b) OR c), reset) -> c.
     OUTPUT (a, b, c)
  END.
```

10.4.2.2 *BKE* Description

The ELLA-text would then be translated into *BKE*:

```
fn BOOL˙DELAY(x) =
  ldelay(x, 0, 1),

fn AND(x, y) =
 ¬(¬ x + ¬ y),
```

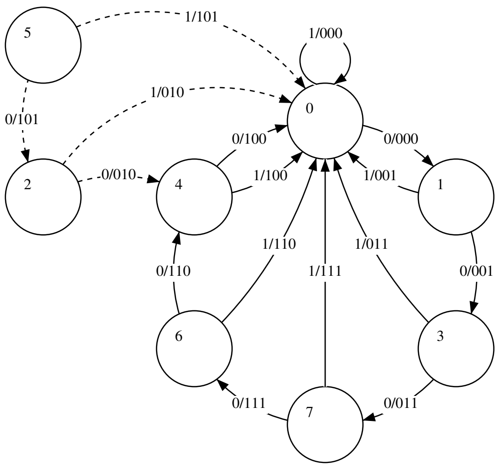

Figure 10.4: IO Automaton for Twisted Ring-Counter

```
fn LATCH(i, r) =
  BOOL_DELAY(AND(i, ¬ r)),

fn TWIST(reset) =
  let  a = LATCH(b, reset),
       b = LATCH(c, reset),
       c = LATCH(AND(¬ a, ¬ b + c), reset)
  in (a, b, c).
```

Note that if $reset = 0$, then the counter will cycle through the six valid states; if $reset = 1$ then state 0 is entered and (0,0,0) will be output at the next time unit. If the reset is always raised, then the counter always outputs (0,0,0), after the first time step, and remains in state 0. Whatever the initial state, or value on *reset*, the counter will output (0,0,0) an arbitrary number of times. However, it is not the case that the counter will output any other value an arbitrary number of times; for example it will only output (1,1,1) at most once if *reset* is always held high. This is the case for any output value except (0,0,0). These two properties will be verified in chapter 12.

10.4.2.3 IO Automaton Description

An IO automaton is now constructed from the *BKE* description of the counter. In fact, it is isomorphic up to state names with that of Figure 10.4. As may be expected, the automaton has eight states; as before, they have been given labels which happen to correspond to the integer values of the boolean output produced when leaving each state. Again, the state labels do not have any further significance.

The automaton shown in Figure 10.4 is constructed according to the semantic rules for *BKE* described in §6.3, by composition of the automata representing the sub-components of each unit. The label 'r/xyz' on a transition arc means that if the input has value $reset = r$ then the transition can take place, producing an output for the counter of $a = x,\ b = y$ and $c = z$, where the outputs of the counter are labelled a, b and c.

This process is now described, and the transition tables for these intermediate automata are given.

The function object for *LATCH* has an argument count of 2, and a two-state automaton whose transition set is given by the following table. We have labelled the states 0 and 1. The initial state is state 0. Note that the output value for each transition is the same as the label on the *from* state. This functional dependence of the next output on the state is a consequence of the fact that the outermost operation is a *DELAY*.

From	Input		Output	To
	1	2	1	
0		1	0	0
1		1	1	0
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0
1	1	0	1	1

The three equations in the final binding for the ring counter give rise to three automata, whose causal product is the automaton for the ring counter. We will construct the product in a single step, using the fact that the product operation is associative up to isomorphism of automata.

The automaton for the equation $a = LATCH(b, reset)$ is obtained from the *LATCH* automaton by simply renaming the inputs and outputs. The initial state is still state 0.

From	Input		Output	To
	b	reset	a	
0		1	0	0
1		1	1	0
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0
1	1	0	1	1

The automaton for the equation $b = LATCH(c, reset)$ is obtained similarly to the above.

From	Input		Output	To
	c	reset	b	
0		1	0	0
1		1	1	0
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0
1	1	0	1	1

The equation for c is more complicated, since c occurs on both left and right hand sides. We proceed in three stages.

First, the expression $AND(\neg a, \neg b + c)$ contains no *DELAY* operators, and so gives a one-state automaton. The transition table is given below.

Input			Output
a	b	c	1
1			0
0	0		1
0	1	0	0
0	1	1	1

Substituting the above table into the table for the *LATCH* automaton and renaming input 2 to *reset* (which amounts to evaluating the function object with the correct arguments) gives the following two-state automaton for the expression $LATCH(AND(\neg a, \neg b + c), reset)$.

From	Input				Output	To
	a	b	c	reset	1	
0				1	0	0
1				1	1	0
0	1			0	0	0
0	0	0		0	0	1
0	0	1	0	0	0	0
0	0	1	1	0	0	1
1	1			0	1	0
1	0	0		0	1	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1

Next, we must rename the output of this automaton to c. In doing so, we must suppress the input c and exclude all transitions in which the values of c and 1 disagree. We then obtain the following automaton for the equation $c = LATCH(AND(\neg a, \neg b + c), reset)$. Once again, the initial state is 0.

From	Input			Output	To
	a	b	reset	c	
0			1	0	0
1			1	1	0
0	1		0	0	0
0	0	0	0	0	1
0	0	1	0	0	0
1	1		0	1	0
1	0	0	0	1	1
1	0	1	0	1	1

The automaton for the ring counter is obtained by taking the causal product of the automata for a, b and c. The calculation is greatly simplified by the functional dependence of the output values on the *from* state, which carries over into the product. The state set is now $\{0,1\}^3$, with the order on the factors given by (a,b,c). The initial state is $(0,0,0)$.

From	Input	Output			To
	reset	a	b	c	
(0, 0, 0)	0	0	0	0	(0, 0, 1)
(0, 0, 0)	1	0	0	0	(0, 0, 0)
(0, 0, 1)	0	0	0	1	(0, 1, 1)
(0, 0, 1)	1	0	0	1	(0, 0, 0)
(0, 1, 0)	0	0	1	0	(1, 0, 0)
(0, 1, 0)	1	0	1	0	(0, 0, 0)
(0, 1, 1)	0	0	1	1	(1, 1, 1)
(0, 1, 1)	1	0	1	1	(0, 0, 0)
(1, 0, 0)	0	1	0	0	(0, 0, 0)
(1, 0, 0)	1	1	0	0	(0, 0, 0)
(1, 0, 1)	0	1	0	1	(0, 1, 0)
(1, 0, 1)	1	1	0	1	(0, 0, 0)
(1, 1, 0)	0	1	1	0	(1, 0, 0)
(1, 1, 0)	1	1	1	0	(0, 0, 0)
(1, 1, 1)	0	1	1	1	(1, 1, 0)
(1, 1, 1)	1	1	1	1	(0, 0, 0)

The last table matches the transition set for the IO automaton representing the twisted ring-counter shown in Figure 10.4.

10.4.3 Equivalence of semantic objects

One can see by inspection that the semantic objects denoting the high and low (gate) level *BKE* descriptions of the twisted ring counter are isomorphic, indeed, we have only given one picture of the structure. The isomorphism implies trace equivalence of the two models and hence that the gate level circuit faithfully models the first description.

In practice, it is necessary to apply the algorithm given in section 10.2. This will establish the trace equivalence and containment relationship that holds between the models. Of importance is the fact that the models are isomorphic up to state names, then the algorithm is of linear time complexity.

Chapter 11

ELLA to ELLA Transformation

This chapter will indicate the direction of future work on transformation based verification, due to commence in year two. Broadly, the issues concern the development of equational laws for Kernel ELLA which are demonstrably sound, with respect to the semantics given. Such equational laws can then be used as a sound basis for transforming ELLA texts.

As such, this will form part of further work during the project.

Chapter 12

Verification of Temporal and Ordering Properties

This chapter contains a discussion of verification of behavioural properties described by temporal logic formulae and automata. The notion of IO automata introduced in Chapter 6 is extended to include Büchi style acceptance conditions. The remainder of the chapter discusses verification and includes some examples.

12.1 IO Automata with Acceptance Conditions

In the previous chapters we have been interested in all possible runs of an IO automaton, i.e. all finite and infinite runs. By being more restrictive on the runs of an automaton, it is possible to obtain a strong expressiveness relationship between IO automata and propositional temporal logic; it then becomes possible to use such enriched structures to characterise temporal, or ordering, properties. In particular, we need to be able to restrict the number of finite runs that are acceptable and also the number of infinite runs that are acceptable. This is achieved by including *terminal* and *acceptance* sets. The terminal set is used to determine acceptable finite runs; the acceptance set is used to determine the acceptable infinite runs.

Formally, we extend the definition of section 6.2 as below.

$$\begin{array}{lrcl}
\mathit{Trans} :: & \mathit{from} & : & \mathit{State} \\
& \mathit{ivals} & : & \mathit{Name} \xrightarrow{m} \mathit{Val} \\
& \mathit{ovals} & : & \mathit{Name} \xrightarrow{m} \mathit{Val} \\
& \mathit{to} & : & \mathit{State}
\end{array}$$

$$\begin{array}{lrcl}
\mathit{IOA} :: & \mathit{states} & : & \mathit{State}\textbf{-set} \\
& \mathit{inputs} & : & \mathit{Name}\textbf{-set} \\
& \mathit{outputs} & : & \mathit{Name}\textbf{-set} \\
& \mathit{transitions} & : & \mathit{Trans}\textbf{-set} \\
& \mathit{start} & : & \mathit{State}\textbf{-set} \\
& \mathit{finish} & : & \mathit{State}\textbf{-set} \\
& \mathit{acceptance} & : & \mathit{State}\textbf{-set-set}
\end{array}$$

$$inv\text{-}IOA : State\text{-}\mathbf{set} \times Name\text{-}\mathbf{set} \times Name\text{-}\mathbf{set} \times Trans\text{-}\mathbf{set} \times State\text{-}\mathbf{set} \times State\text{-}\mathbf{set} \times State\text{-}\mathbf{set}\text{-}\mathbf{set} \to \mathbb{B}$$

$$inv\text{-}IOA(ss, i, o, ts, st, fi, ac) \quad \triangleq \quad st \subseteq ss \wedge fi \subseteq ss \wedge \bigcup ac \subseteq ss \wedge i \cap o = \{\,\} \wedge$$
$$\forall t \in ts \cdot from(i) \in ss \wedge to(i) \in ss \wedge \mathbf{dom}\, ivals(t) = i \wedge \mathbf{dom}\, ovals(t) = o$$

12.1.1 Runs of an automaton

A run rn of an IO-automaton A is a sequence of transitions such that, for all $k, (k+1) \in \mathbf{inds}\, rn$):

(a) $rn[k] \in transitions(A)$
(b) $to(rn[k]) = from(rn[k+1])$

Notice that the run sequence rn may be finite or infinite. We define the set of runs for an IO automaton A to be $Run(A)$ where:

$$Run(A) = \{rn \in transitions(A)^{\infty} \mid rn \text{ is a run of } A\}$$

and, within a run, rn, the set of states that occurs *infinitely often*:

$$inf(rn) = \{s \in states(rn) \mid \forall\, i \in \mathbb{N} \bullet\ \exists\, j \in \mathbb{N} \bullet\ i < j \wedge s = start(rn[j])\}$$

and where $states(rn) = \{from(tn), to(tn) \mid tn \in \mathbf{elems}\, rn\}$.

Using these concepts we can define the notion of *acceptance* for the automaton A. We say that $rn \in Run(A)$ is an *accepting run* of A iff:

(a) $from(first(rn)) \in start(A)$
(b) for finite rn, $to(last(rn)) \in terminal(A)$
(c) for infinite rn, $\exists ac\text{-}st \in acceptance(A) \bullet\ ac\text{-}st \subseteq\ inf(rn)$

where $first(rn)$ and $last(rn)$ gives the first and last transition of the run rn respectively. The first two parts of this definition are standard from automata theory; however, the third part is a form of *generalised Büchi* acceptance condition which allows the automaton to discriminate on the infinite words that it accepts. It says that an infinite run is accepted whenever the set of all states appearing infinitely often within the run contains an accepting subset (i.e. an element of $acceptance(A)$). This condition is easily shown to be as expressive as the original Büchi acceptance condition[Tho90] - in some sense, this is *dual* to the form of generalised Büchi acceptance condition [VW88, Wol89]. In Wolper's original form of this condition, a non-emptiness check is made with every set in the set of accepting subsets i.e. the condition is *conjunctive* whereas the one mentioned above is *disjunctive*.

We define the set of all accepting runs of automaton A to be denoted by $AccRun(A)$.

As a special case, note that if $terminal(A) = states(A)$ and $acceptance(A) = \{\emptyset\}$ then every finite or infinite run starting in an initial state is an accepting run. Moreover, if $acceptance(A) = \emptyset$ then A can only accept finite runs; conversely, if $terminal(A) = \emptyset$ then A can only accept infinite runs. Obviously, if both $terminal(A)$ and $acceptance(A)$ are empty, then A has no accepting runs at all.

12.1.2 Language acceptance by IO automata

The notions of language acceptance or recognition by an automaton are easily defined in terms of accepting runs. The *input/output language* accepted/generated by automaton A is defined to be $IOLang(A)$, where:

$$IOLang(A) = \{iotrace(rn) \in IOSig^{\infty} \mid rn \in AccRun(A)\}$$

where

$$\begin{aligned} iotrace(rn)[i] &= \big(ivals(rn[i]), ovals(rn[i])\big) \\ IOSig &= (ISig \times OSig) \\ ISig &= (Name \xrightarrow{m} Val) \\ OSig &= (Name \xrightarrow{m} Val) \end{aligned}$$

Similarly, for an automaton A, we can define the *input language* that it *accepts* and the *output language* that it *generates*; call these $ILang(A)$ and $OLang(A)$ respectively, where:

$$ILang(A) = \{itrace(rn) \in ISig^{\infty} \mid rn \in AccRun(A)\}$$

$$OLang(A) = \{otrace(rn) \in OSig^{\infty} \mid rn \in AccRun(A)\}$$

where

$$\begin{aligned} itrace(rn)[i] &= ivals(rn[i]) \\ otrace(rn)[i] &= ovals(rn[i]) \end{aligned}$$

For later work, we need the following relative IO language for a given automaton A and subset J from the input signature, $ISig$:

$$IOLang(A,J) = \{iotrace(rn) \in IOSig^{\infty} \mid rn \in AccRun(A) \land \ itrace(rn) \in J^{\infty}\}$$

12.1.3 Trace containment and equivalence

Using these notions of language acceptance, we continue by redefining the notion of *trace containment* and *trace equivalence*. These will play an important rôle in verification, implementation and refinement. Let A_1 and $A_2 \in IOA$, then:

A_1 is trace contained by $A_2 \;\Leftrightarrow\; IOLang(A_1) \subseteq IOLang(A_2)$

A_1 is trace equivalent to $A_2 \;\Leftrightarrow\; IOLang(A_1) = IOLang(A_2)$

Symbolically, we write $(A_1 \subseteq_{\mathrm{t}} A_2)$ to mean that the traces of A_1 are contained in the traces of A_2 and also that $(A_1 \equiv_{\mathrm{t}} A_2)$ to mean that the traces for A_1 and A_2 are identicle.

12.1.4 Implementation relation

We now redefine what it means for the behaviour of one automaton A_1 to be *implemented* by the behaviour of another automaton A_2. The idea is that A_2 can accept a wider input language than A_1, while also ensuring that A_2 performs the same output actions on inputs common to both A_1 and A_2. More formally, we say that:

A_1 is implemented by $A_2 \;\Leftrightarrow\; IOLang(A_2, ILang(A_1)) = IOLang(A_1)$

Symbolically, we write $(A_1 \sqsubseteq_{\mathrm{i}} A_2)$ whenever A_1 is implemented by A_2. Note that $A_1 \sqsubseteq_{\mathrm{i}} A_2$ implies that $ILang(A_1) \subseteq ILang(A_2)$.

12.2 Verification of Trace Containment

Given two automata A_1 and A_2 we can determine whether A_1 is trace contained by A_2 by establishing whether $\mathit{IOLang}(A_1) \cap \overline{\mathit{IOLang}(A_2)} = \emptyset$. The complement of the language of an IO automaton can be obtained by taking the language of the "complement" of the IO automaton itself. Furthermore,

$$(\mathit{IOLang}(A_1) \cap \mathit{IOLang}(A_2)) = \mathit{IOLang}(A_1 \times A_2).$$

The product operation on automata is a simple operation. So provided we can complement an IO automaton and can test for language emptiness of IO automaton, we have a route for verification. Unfortunately, performing the complementation of (non-deterministic) IO automata is, in general, a rather complex task, so can the complementation on IO automata be avoided? If we wish to determine whether a temporal logic property holds on an automaton, then the answer is clearly yes. One can build an automaton that corresponds to the logically negated property. What about testing for emptiness of an IO automaton? Fortunately, straightforward algorithms exist for that too.

12.2.1 Algorithm for Language Emptiness of an IO Automaton

Here we outline, informally, such an algorithm[1]. Considering the definition of acceptance of a run of IO automaton given in the above section 12.1.1, notice that there are essentially two cases to consider. An automaton will be empty if there are no finite runs that start in an initial state and there are no infinite runs that start in an initial state.

For the finite run case, we simply determine, for each terminal state appearing in the automaton state set, whether it is connected to some initial state. If none are connected (via the transition relation), then clearly there are no finite runs.

The infinite run case is somewhat more complicated. It is more convenient to think of the automaton as a graph, whose nodes are the states and edges are the transitions. If an infinite run exists in the automaton, then there must be a maximal strongly connected component, MSCC, within the state graph of the automaton which contains the infinity set of the run, and is connected to an initial state. Furthermore, if the infinite run is *acceptable*, then one of the acceptance subsets of the automaton must be a subset of the states of the particular MSCC. This therefore gives an algorithm. We can compute the MSCCs of a graph in linear time — that is Tarjan's algorithm — then check each for containment of an accepting subset. If one is found, then determine whether it is connected to an initial state — if it is, the automaton will not be empty.

12.3 Example: Verifying temporal properties of the TRC

Let us now turn our attention to the verification of temporal, or ordering, properties of a *BKE* description. For convenience, we'll continue with the twisted ring counter example of chapter 10. For example, we should be able to establish that, no matter how reset behaves, the counter will output $(0,0,0)$ infinitely often. As indicated above, we can establish such temporal properties by first computing an automaton that denotes the complement of the property, then computing the (standard) product and checking whether the result is empty, i.e. denotes an empty language. However, to represent such temporal properties, the automata must make use of the Büchi-like acceptance condition. Figure 12.1 depicts the structure representing the complement of the above property. The double arrows in these diagrams are used to represent sets of transitions, all having the same input conditions and output values. The transition labelling

[1]In later version of this chapter, a more formal presentation will be provided.

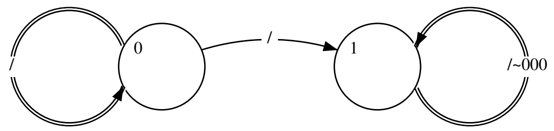

Figure 12.1: IO Automaton for complement of property

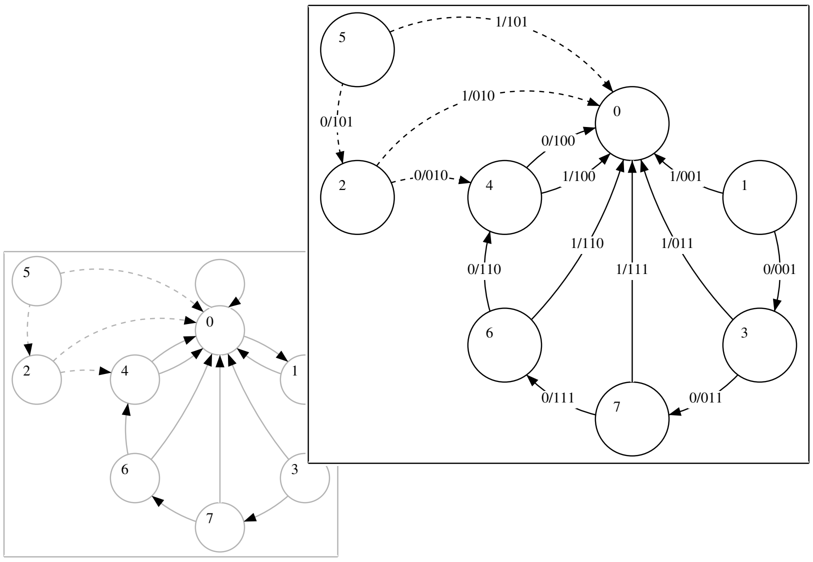

Figure 12.2: Product IO Automaton to test temporal property

has the same meaning as before; for example '/' means that a transition can occur for any value of *reset* producing any value for (a,b,c), and '$/\neg000$' means that a transition can occur for any value of *reset* producing any value *except* (0,0,0) for (a,b,c).[2] The initial state is 0, the terminating set is empty, and the accepting set is $\{\{1\}\}$, i.e. all infinite runs must visit state 1 infinitely often, alternatively there are no infinite runs that remain in state 0. The product between this automaton and the IO automaton for the twisted-ring counter of Figure 10.4 is then determined and shown in Figure 12.2. This automaton is depicted in two layers: the bottom layer, not completely shown, consists of that part of the product which comes from combining state 0 of the complement property automaton with the counter structure; the upper layer comes from combining state 1 with the counter structure. Each state in the lower layer has transitions to its "counting" successors in both layers. However, the states in the upper layer have no

[2]The IO automata representing this property and its complement may be expressed initially using a temporal logic query language over the IO automaton transitions. For example, if 'i/$\neg a \wedge \neg b \wedge \neg c$' is taken to mean that a transition will occur with output $a=0$, $b=0$ and $c=0$, for an input with value i, then the mentioned property would be expressed as $\Box\Diamond(/\neg a \wedge \neg b \wedge \neg c)$, with its complement '$\Diamond\Box\neg(/\neg a \wedge \neg b \wedge \neg c)$'. The temporal operators '$\Box$' and '$\Diamond$' have their usual meanings of 'Always' and 'Eventually', respectively.

transitions exiting from the “initial” state, labelled 0. Only upper layer states are accepting. Since there are no cycles in the upper layer, there can be no runs visiting upper layer states infinitely often. Hence the automaton denotes an empty language and thus the original property is established. Again, standard algorithms are used to detect this emptiness.

Chapter 13

Test Vector Generation and Simulation

This chapter will contain a discussion of the generation of test vectors from semantics and the use of the IOA semantics as a simulator generator, highlighting the differences between this and the standard ELLA simulator.

As such, this will form part of further work during the project.

Chapter 14

Abstraction, Hiding and Composition

This chapter will contain material on abstraction, hiding and composition. These issues need to be addressed in order to promote the hierarchical, compositional verification of modular components, and thereby encouraging their reuse.

As such, this will form part of further work during the project.

Chapter 15

Representation Issues

This chapter contains a discussion of how the use of appropriate representation techniques for IO automata may be used to reduce (space) complexity.

IO Automata: A Mealy style automaton model in which information is carried by the transition arcs, rather than being explicitly embedded within states. Each transition arc is a directed link from one state to another and has a label structure consisting of a pair containing the input and output information. Even by itself, this generally reduces the number of states required to represent systems when compared to Moore machine models, whilst potentially increasing the number of transitions.

It is presently unclear by how much the number of transitions might increase in specific cases when going from a Moore-style automaton to an IO automaton. The examples we have looked at so far have shown little increase and sometimes a decrease. However, note that the maximum number of transitions is bounded above by the product of number of states with the total number of possible input and output mappings. Hence, if the size of state space decreases then this upper bound decreases also.

Partial mappings: We are able to use partial mappings in the transitions to record when the inputs or outputs do not care how particular labels are to be interpreted. Therefore, this provides one natural and compact way to bundle together transitions. Since the input and output signatures and the value sets are known for any IO automaton, any such partial mapping represents the set of all total mappings on the appropriate signature that are functionally consistent with the given partial mapping.

Intensional representations: This covers *symbolic* techniques of representing information which would be otherwise highly inefficient to represent in an extensional way i.e. by means of an explicit table or enumeration.

Bundling up transitions: The use of transitions to carry information rather than in the states permits the following observation. It enables transitions with the same start and finish states to be bundled together and then treated as functions, at least in the deterministic case. This introduces the potential for introducing symbolic representations for bundled transitions.

Using predicates for sets: There are various places where sets are required in the definition of IO automata, such as in the initial states, the terminal states and the accepting states. It would then be possible to represent such sets by means of a logical predicate, rather than by a tedious and space consuming enumeration of the elements.

However, although these intensional descriptions are often compact, their main advantage is also a serious disadvantage: they are essentially *syntactic programs* which need further processing and interpretation to yield any semantic information. If they are not constrained in some way, they simply do not advance the analysis process, but merely *defer* it to the point where expansion to the corresponding extension becomes inevitable. In brief, the problem with intensional representation is that their expressiveness is too easily increased to the point where they become very hard or impossible to analyse *decidably*.

Hence, the main issue here is to carefully consider the class of symbolic expressions used for bundling transitions and representing predicates to ensure that the appropriate symbolic analysis remains *tractable* and so decidable. It is also clear that this work on symbolic manipulation has a very natural tie-up with forthcoming work in ELLA transformation later on the project.

Appendix A

Verification requirements

A.1 Summary of Designer Requirements

The following list is a summary of Designer Requirements [i]-[xiii] identified at the industrial sites. There is a reference to the Verification Requirements ([A]-[C],[1]-[16]) which coincide with each Designer Requirement; these are described below in §A.2. If these Verification Requirements are met by a future design system then these corresponding Designer Requirements will also be met.

[i] support tools for reliable design translation (including datatype, level and timing translations) (met by Verification Requirements [A], [B], [C]; [4], [8])

[ii] test pattern comparison tool (to cope with rescheduling problems) (met by [4(b)])

[iii] high-level schematic capture, but with a sketch-pad feel for flexible use (for example, to cope with: varying degrees of detail; incomplete and inconsistent designs; (which would be worked on at a later date)) (met by [5])

[iv] hierarchical timing analyser to bridge gap between ELLA simulator (with no real-time consideration) and present timing analysers (which flatten design) applied to standard cells (met by [12])

[v] design management system/support (for example, code management, design route standardisation, design partition) (met by [7(b)])

[vi] methodology to construct abstract model which facilitates mapping to low-level physical implementation, especially with respect to timing. (met by [A], [B], [C], particularly [3])

[vii] behavioural/functional specification for block interfaces (met by [7(d)])

[viii] Interface with specialist simulation/CAD tools.

[ix] block diagram *output* from initial ELLA-text, so that dataflow can be observed (a mixture of text and picture would be good) (met by [5])

[x] more degrees of abstraction; do not need actual physical timing, just consider architectural aspects. (met by [7], [2])

[xi] behavioural synthesis tool

[xii] adaptable ELLA environment user-interface.

[xiii] dynamic storage in future versions of ELLA

A.2 Summary of Verification Requirements

The requirements for the following three distinct design verification areas must be considered (only for [B] is it possible to use purely formal techniques).

[A] User Requirements vs. User Specification

[B] Transformations applied to Abstract Design Representation; for example:

- (i) Specification A vs. Specification B
- (ii) Specification vs. Draft Design
- (iii) Draft Design A vs. Draft Design B

[C] Implementation Design vs. Physical Device

Note that for each Verification Requirement [1]-[16], the corresponding Designer Requirements [i]-[xiii] which will be fulfilled have been noted:
[A] User Requirements vs. User Specification (see [SBE90]); partly meets Designer Requirements [i], [vi]:

[1] *Formalised* Specification Language:

[2] Properties of Specification language: simplicity, ‘natural’ hardware description, (de)composition, (data/timing) abstraction (meets [x])

[3] Well-defined design process (especially during capture of user requirements) Iterative/consultative construction of User Specification in Exploration Phase of Design Process Abstraction Phase to remove unnecessary design details, such as implementation dependencies (partly meets [vi])

[4] Animation of User Specification (partly meets [i])

[5] Schematic capture/animator, timing diagram processor; Documentation/Text/picture processing/annotation/hyper-linking with (formal) specification and rest of design environment (meets [iii], [viii])

[6] Designer Education

[B] User Specification vs. Implementation Design (partly meets [i], [vi]):

[4] Animation (and suitable animation processing tools)

[7] General model/language properties (partly meets [x]):

- (a) formal model/language, with many-sorted ADTs, multiple clocks
- (b) well-defined design process, with design/code management (meets [v])
- (c) Abstraction (data, control, timing)

(d) Composition or decomposition (meets [vii])

(e) machine-processable syntax/semantics

[8] Formal Verification (partly meets [i]):

(a) equivalence checking

(b) property checking (invariants, conditions, eventualities)

(c) theorem proving
(This needs formal model, with abstraction, composition, retiming)

[9] realistic computational models (especially at low level)

[C] Implementation Design vs. Physical Device:

[10] Testing methodology (to test for fabrication faults *and* for functionality faults).

[11] experiments to demonstrate that lowest level design does not *violate* product design

[12] Realistic simulation and timing analysis (meets [iv]):

(a) multi-level

(b) compatible with physical device behaviour

(c) compatible with low level discrete model of design

[14] experimental results

[15] links between simulators and discrete design representation

[16] Electronic/physics theoretical development of semi-conductor circuit modelling techniques.

The need for animation introduces further requirements on the design environment:
[4] Animation:

(a) executable specifications

(b) test pattern comparison tool (meets [ii]):

(i) test pattern generator

(ii) retiming/rescheduling tool

(iii) datatype conversion tool

(iv) like-data-comparison tool

(c) multi-level

[7(c)] Abstraction:

(i) Data abstraction: many-sorted ADTs, homomorphisms, compilers, ADT extensions; separate system Data from Control

(ii) Timing abstraction: multiple clocks, and maps between them (retiming); adoption of synchronous model

A.3 Glossary

- *User*: the person, group or company who will use a fabricated device. They provide a set of requirements (behavioural, physical) which they wish the device to fulfil.
- *Designer*: the person or group of engineers who specify, design and implement a device, given the user requirements.
- *User Concepts* or *User Requirements*: information provided by the user to describe the behaviour of the device they want; typically this will be an ad hoc textual/pictorial description, algorithm description, interface requirements, including operating constraints (area, speed, power etc); mathematical formulae, timing diagrams, schematics.
- *User Specification*: the capture of user requirements in a formal specification language/model; needs consultation with user, animation, abstraction, iteration; there will only be an informal correspondence between user requirements and user specification
- *Hardware Verification*: determining whether a physical computing device exhibits the behaviour required by the device User.
- *Formal Verification*: determining, using formally-based techniques, that an abstract mathematical model of a computing device possesses a set of properties, and whether two abstract models are equivalent.
- *(traditional) CAD tools*: software to assist a designer in producing graphical descriptions of a VLSI circuit, and in checking that the circuit will operate correctly. For example: graphics input/editing packages, (layout) design-rule checkers, electrical rule checkers, circuit extractors, logic-level simulators, timing analysers, circuit-level simulators.
- *(Abstract) Data types (ADTs)*: can be defined in terms of a many-sorted algebra, which consists of a collection of non-empty sets of elements, with operations between them. An ADT is defined as the *isomorphism class* of a many-sorted algebra.
- *Schematic Capture*: entering circuit, manipulating it symbolically; i.e., draw circuits/wires, blocks/busses, annotate, and then process them using other design tools.
- *Simulation*: trying to imitate the behaviour of a device using model of the design
- *Animation*: applying inputs to a *model*, using it to perform *its* computation and observing the output of the model (so an animation may have nothing to do with the behaviour of any *real* computing device, whereas a simulation does try to reproduce the potential behaviour of the device)

Appendix B

ELLA Composite Syntax

This appendix presents the ELLA V6 Syntax[1]

B.1 Basic Notation

Throughout this appendix the following notation will be used

abc $\in$ Abc	$\equiv$	'abc' is an element of the set 'Abc'
b ::= c	$\equiv$	the syntax definition of 'b' is 'c'
|	$\equiv$	the separator of alternatives in a syntax definition
$\vert$	$\equiv$	ELLA separator of alternatives
int_string	$\equiv$	an ELLA integer name or string of printable characters
$d_1 \cdots d_k$	$\equiv$	one or more occurrences of 'd'
$d_1, \cdots, d_k$	$\equiv$	one or more occurrences of 'd' separated by ','. Note if k=1 then no ',' is present.
$d_1, \cdots, d_{k-1}$	$\equiv$	zero or more occurrences of 'd' separated by ','. Note if k=0 then no ',' is present.

B.2 Conventions

typename	$\in$	Identifier	(ELLA type name i.e. lower case)
intname	$\in$	Identifier	(Integer name i.e. lower case)
constname	$\in$	Identifier	(ELLA constant name i.e. lower case)
signalname	$\in$	Identifier	(ELLA signal name)
tagname	$\in$	Identifier	(ELLA tagged type name)
altname	$\in$	Identifier	(ELLA enumerated type alternative)
macintname	$\in$	Identifier	(Macro integer parameter name)
mactypename	$\in$	Identifier	(Macro type parameter name)
repname	$\in$	Identifier	(Replicator variable name)
nullname	$\in$	Identifier	(An unnamed identifier, given by a blank space)
attribute_name	$\in$	Identifier	(An attribute identifier name)
fnname	$\in$	Fnname	(ELLA function name i.e. upper case or symbol)
biopname	$\in$	Fnname	(ELLA BIOP name i.e. upper case)

[1]Crown Copyright ©Controller HMSO, London, 1991.

z	$\in$	$\mathbb{Z}$	(An integer)
j,k	$\in$	$\mathbb{N}_1$	(A non-zero positive integer)
ei	$\in$	Integer	('ei' is a value of the ELLA Integer type 'Integer')
index	$\in$	Integer	(index of structure)
lwb, upb	$\in$	Integer	(Lower-bound and Upper-bound of a range)
size	$\in$	Integer	(Size of row or string)
interval	$\in$	Integer	(ELLA timing interval)
ambig_time	$\in$	Integer	(Ambiguity delay time)
delay_time	$\in$	Integer	(delay time)
skew_time	$\in$	Integer	(skew time value of input data)
slow_time	$\in$	Integer	(slow down factor)
fast_time	$\in$	Integer	(speed up factor)
con	$\in$	Constant	('con' is a value of the ELLA Constant type 'Constant')
initial_value	$\in$	Constant	(Delay, Retiming or Ram initialisation value)
ambig_value	$\in$	Constant	(Delay ambiguity value)
char	$\in$	Character	(A printable character i.e. 'a')
firstchar	$\in$	Character	(First printable character in a range)
lastchar	$\in$	Character	(Last printable character in a range)
string	$\in$	String	(A string of printable characters i.e. 'abc')

B.3 Syntactic Definitions

Enumerated

enumerated	::=	altname
	|	tagname / integer
	|	tagname $'$char
	|	tagname $''$string$''$

Integer

integer	::=	integer DOP integer1
	|	INT macintname$_1$ $\cdots$ macintname$_k$
	|	integer1
integer1	::=	integer1 + integer2
	|	integer1 - integer2
	|	integer2
integer2	::=	integer2 $*$ integer3
	|	integer2 % integer3
	|	integer3
integer3	::=	z

| intname
| MOP integer3
| IF bool THEN integer ELSE integer FI
| (integer)

DOP ::= SL
| SR
| IAND
| IOR
| MOD

MOP ::= +
| -
| INOT
| ABS
| SQRT

Bool

bool ::= integer BOP integer
| NOT bool

BOP ::= =
| / =
| >
| <
| >=
| <=
| AND
| OR

Type

type ::= () → type
| type → ()
| type → type
| type1

type1 ::= typename
| STRING [size] type1
| [size] type1
| (type_1 , $\cdots$, type_k)
| TYPE $\text{mactypename}_1 \cdots \text{mactypename}_k$

Constant

```
const          ::=  STRING [ size ] const1
                 |  [ size ] const
                 |  const1

const1         ::=  constname
                 |  enumerated
                 |  tagname & const1
                 |  tagname ( 'firstchar .. 'lastchar )
                 |  tagname / ( lwb .. upb )
                 |  ( constset1 , ··· , constsetk )
                 |  ? type1
                 |  type1
```

Constset

```
constset       ::=  const1  |···  |constk
```

Unit

```
unit           ::=  unit CONC unit1
                 |  unit fnname fnparams attributes unit1
                 |  unit1

unit1          ::=  STRING [ size ] unit1
                 |  [ size ] unit1
                 |  [ INT repname = lwb .. upb ] unit1
                 |  fnname fnparams attributes unit1
                 |  fnname fnparams attributes ()
                 |  tagname & unit1
                 |  unit2 // tagname
                 |  unit2 attributes

unit2          ::=  signalname
                 |  enumerated
                 |  IO signalname
                 |  unit2 [ index ]
                 |  unit2 [ lwb .. upb ]
                 |  unit2 [[ unit ]]
                 |  IF bool THEN unit ELSE unit FI
                 |  ? type1
                 |  closedclause

fnparams       ::=  { params1 , ··· , paramsk }
                 |  nullname

params         ::=  integer
                 |  type
                 |  const
                 |  fnname
                 |  macname fnparams
```

attributes ::= attname$_1$ $\cdots$ attname$_{k\text{-}1}$

attname ::= @ attribute_name

Closedclause

closedclause ::= CASE unit OF cases elsecases ELSE unit ESAC
| CASE unit OF cases elsecases ESAC
| (unit$_1$, $\cdots$, unit$_k$)
| BEGIN step$_1$ $\cdots$ step$_{k\text{-}1}$ OUTPUT outputunit END
| BEGIN step$_1$ $\cdots$ step$_k$ END
| (step$_1$ $\cdots$ step$_{k\text{-}1}$ OUTPUT outputunit)
| (step$_1$ $\cdots$ step$_k$)
| BEGIN SEQ sequence$_1$; $\cdots$; sequence$_{k\text{-}1}$; OUTPUT outputunit END
| BEGIN SEQ sequence$_1$; $\cdots$; sequence$_{k\text{-}1}$ END
| (SEQ sequence$_1$; $\cdots$; sequence$_{k\text{-}1}$; OUTPUT outputunit)
| (SEQ sequence$_1$; $\cdots$; sequence$_{k\text{-}1}$)

outputunit ::= unit
| ()

cases ::= constset$_1$: unit$_1$, $\cdots$, constset$_k$: unit$_k$

elsecases ::= elseof$_1$ $\cdots$ elseof$_{k\text{-}1}$

elseof ::= ELSEOF cases

step ::= declarations .
| LET letdecs$_1$, $\cdots$, letdecs$_k$.
| MAKE makedecs$_1$, $\cdots$, makedecs$_k$.
| JOIN joindecs$_1$, $\cdots$, joindecs$_k$.
| FOR multiplejoin$_1$ $\cdots$ multiplejoin$_k$ JOIN joindecs$_1$, $\cdots$, joindecs$_j$.
| PRINT printitem$_1$, $\cdots$, printitem$_k$.
| FAULT printitem$_1$, $\cdots$, printitem$_k$.

declarations ::= INT intdec$_1$, $\cdots$, intdec$_k$
| TYPE typedec$_1$, $\cdots$, typedec$_k$
| CONST constdec$_1$, $\cdots$, constdec$_k$
| FN fndec$_1$, $\cdots$, fndec$_k$
| MAC macdec$_1$, $\cdots$, macdec$_k$

decnames ::= signalname
| (multiplenames$_1$, $\cdots$, multiplenames$_k$)

multiplenames ::= signalname
| nullname

letdecs	::=	decnames = unit
makedecs	::=	fnname fnparams fnparams attributes : signalname$_1$ $\cdots$ signalname$_k$
	|	[size] makedecs
joindecs	::=	unit $\rightarrow$ joinunit
joinunit	::=	joinunit CONC joinunit1
	|	joinunit1
joinunit1	::=	[INT repname = lwb .. upb] joinunit1
	|	joinunit2
joinunit2	::=	signalname
	|	IO signalname
	|	joinunit2 [lwb .. upb]
	|	joinunit2 [index]
	|	(joinunit$_1$, $\cdots$, joinunit$_k$)
multiplejoin	::=	INT repname = lwb .. upb
printitem	::=	IF bool THEN int_string$_1$ $\cdots$ int_string$_k$ FI
	|	int_string$_1$ $\cdots$ int_string$_k$
sequence	::=	declarations
	|	LET letdecs$_1$, $\cdots$, letdecs$_k$
	|	VAR vardecs$_1$, $\cdots$, vardecs$_k$
	|	PVAR pvardecs$_1$, $\cdots$, pvardecs$_k$
	|	PRINT printitem$_1$, $\cdots$, printitem$_k$
	|	FAULT printitem$_1$, $\cdots$, printitem$_k$
	|	sequence2
sequence2	::=	assign := unit
	|	(multassign$_1$, $\cdots$, multassign$_k$) := unit
	|	CASE unit OF seqcase seqelseof ELSE sequence2 ESAC
	|	CASE unit OF seqcase seqelseof ESAC
	|	IF bool THEN sequence2 ELSE sequence2 FI
	|	IF bool THEN sequence2 FI
	|	[INT repname = lwb .. upb] sequence2
	|	(sequence2$_1$; $\cdots$; sequence2$_k$)
seqcase	::=	seqstep$_1$, $\cdots$, seqstep$_k$
seqstep	::=	constset : sequence2
	|	constset :
seqelsecase	::=	seqelseof$_1$ $\cdots$ seqelseof$_{k\text{-}1}$

seqelseof	::=	ELSEOF seqcase
vardecs	::=	decnames := unit
pvardecs	::=	decnames **::=** const
assign	::=	signalname
	|	assign [lwb .. upb]
	|	assign [index]
	|	assign [[unit]]
multassign	::=	assign
	|	nullname

Function Body

functionbody	::=	unit
	|	REFORM
	|	BIOP biopname fnparams
	|	ALIEN alienname fnparams
	|	ARITH integer
	|	DELAY delaybody
	|	IDELAY (initial_value , delay_time)
	|	SAMPLE samplebody
	|	FASTER fasterbody
	|	SLOWER slowerbody
	|	RAM (initial_value)
	|	IMPORT
delaybody	::=	(initial_value , ambig_time , ambig_value , delay_time)
	|	(initial_value , ambig_time , delay_time)
	|	(initial_value , delay_time)
samplebody	::=	(interval , initial_value , skew_time)
	|	(interval)
fasterbody	::=	(fnname , fast_time , initial_value , skew_time)
	|	FASTER (fnname , fast_time)
slowerbody	::=	(fnname , slow_time , initial_value , skew_time)
	|	(fnname , slow_time)

Integer Declaration

intdec	::=	intname = integer

Type Declaration

typedec ::= typename = typeornew

typeornew ::= type
| NEW tagname / (lwb .. upb)
| NEW (typealt_1 $|\cdots$ $|\text{typealt}_k$)
| NEW tagname (charange_1 $|\cdots$ $|\text{charange}_k$)

typealt ::= tagname & type1
| altname

charange ::= $'$char
| $'$firstchar .. $'$lastchar

Constant Declaration

constdec ::= constname = constset

Function Declaration

fndec ::= fnname = terminals $\rightarrow$ outtype : functionbody

terminals ::= (terminaltype_1 , $\cdots$, terminaltype_k)
| ()

terminaltype ::= type : $\text{signalname}_1 \cdots \text{signalname}_k$
| type

outtype ::= terminals
| type

Generalized Function Declaration

fndec ::= fnname = outtype : functionbody

Macro Declaration

macdec ::= fnname maclist = terminals $\rightarrow$ outtype : functionbody

maclist ::= { macpram_1 , $\cdots$, macpram_k }
| nullname

macpram ::= INT $\text{intname}_1 \cdots \text{intname}_k$
| TYPE $\text{typename}_1 \cdots \text{typename}_k$
| CONST type : $\text{constname}_1 \cdots \text{constname}_k$
| FN (type) $\rightarrow$ type : $\text{fnname}_1 \cdots \text{fnname}_k$
| FN type : $\text{fnname}_1 \cdots \text{fnname}_k$
| MAC maclist (type) $\rightarrow$ type: $\text{fnname}_1 \cdots \text{fnname}_k$
| MAC maclist type: $\text{fnname}_1 \cdots \text{fnname}_k$

Generalized Macro Declaration

macdec	::=	fnname maclist = outtype : functionbody

Closure

closure	::=	$\text{declarations}_1 \,.\, \cdots \,.\, \text{declarations}_k$.

Appendix C

Core ELLA Composite Syntax

C.1 Basic Notation

Throughout this appendix the following notation will be used

abc $\in$ Abc	$\equiv$	'abc' is an element of the set 'Abc'
b ::= c	$\equiv$	the syntax definition of 'b' is 'c'
$\mid$	$\equiv$	the separator of alternatives in a syntax definition
$\mid$	$\equiv$	ELLA separator of alternatives
$d_1 \cdots d_k$	$\equiv$	one or more occurrences of 'd'
$d_1, \cdots, d_k$	$\equiv$	one or more occurrences of 'd' separated by ','. Note if k=1 then no ',' is present.
$d_1, \cdots, d_{k\text{-}1}$	$\equiv$	zero or more occurrences of 'd' separated by ','. Note if k=0 then no ',' is present.

C.2 Conventions

typename	$\in$	Identifier	(ELLA type name e.g. lower case)
signalname	$\in$	Identifier	(ELLA signal name)
tagname	$\in$	Identifier	(ELLA tagged type name)
altname	$\in$	Identifier	(ELLA enumerated type alternative)
fnname	$\in$	Fnname	(ELLA function name e.g. upper case or symbol)
biopname	$\in$	Fnname	(ELLA BIOP name e.g. upper case)
z	$\in$	$\mathbb{Z}$	(An integer)
lwb, upb	$\in$	$\mathbb{Z}$	(An integer)
j,k	$\in$	$\mathbb{N}_1$	(A non-zero positive integer)
index	$\in$	$\mathbb{N}_1$	(A non-zero positive integer)
size	$\in$	$\mathbb{N}_1$	(A non-zero positive integer)
interval	$\in$	$\mathbb{N}_1$	(ELLA timing interval)
ambig_time	$\in$	$\mathbb{N}_1$	(Ambiguity delay time)
delay_time	$\in$	$\mathbb{N}_1$	(delay time)
skew_time	$\in$	$\mathbb{N}_1$	(skew delay)
con	$\in$	Constant	('con' is a value of the ELLA Constant type 'Constant')
initial_value	$\in$	Constant	(Delay, Retiming or Ram initialisation value)
ambig_value	$\in$	Constant	(Delay ambiguity value)

char	$\in$	Character	(A printable character e.g. ‘a’)
string	$\in$	String	(A string of printable characters e.g. ‘abc’)

C.3 Syntactic Definitions

Enumerated

$$
\begin{array}{lcl}
\text{enumerated} & ::= & \text{altname} \\
 & | & \text{tagname / z} \\
 & | & \text{tagname } {}'\text{char} \\
 & | & \text{tagname } {}''\text{string}''
\end{array}
$$

Type

$$
\begin{array}{lcl}
\text{type} & ::= & \text{typename} \\
 & | & \text{STRING [size] type} \\
 & | & \text{[size] type} \\
 & | & (\ \text{type}_1, \cdots, \text{type}_k\ )
\end{array}
$$

Constant

$$
\begin{array}{lcl}
\text{const} & ::= & \text{STRING [size] const1} \\
 & | & \text{[size] const} \\
 & | & \text{const1}
\end{array}
$$

$$
\begin{array}{lcl}
\text{const1} & ::= & \text{enumerated} \\
 & | & \text{tagname \& const1} \\
 & | & (\ \text{const}_1, \cdots, \text{const}_k\ ) \\
 & | & \text{? type}
\end{array}
$$

Constset

$$
\begin{array}{lcl}
\text{constset} & ::= & \text{constset}_1\ | \cdots\ | \text{constset}_k
\end{array}
$$

$$
\begin{array}{lcl}
\text{constset1} & ::= & \text{STRING [size] constset2} \\
 & | & \text{[size] constset1} \\
 & | & \text{constset2}
\end{array}
$$

$$
\begin{array}{lcl}
\text{constset2} & ::= & \text{enumerated} \\
 & | & \text{tagname \& constset2} \\
 & | & (\ \text{constset}_1, \cdots, \text{constset}_k\ ) \\
 & | & \text{type}
\end{array}
$$

Unit

unit	::=	unit CONC unit1
	|	unit1
unit1	::=	STRING [size] unit1
	|	[size] unit1
	|	fnname unit1
	|	tagname & unit1
	|	unit2 // tagname
	|	unit2
unit2	::=	signalname
	|	enumerated
	|	unit2 [index]
	|	? type
	|	closedclause

Closedclause

closedclause	::=	CASE unit OF cases ELSE unit ESAC
	|	($\text{unit}_1, \cdots, \text{unit}_k$)
	|	BEGIN $\text{step}_1 \cdots \text{step}_{k\text{-}1}$ OUTPUT unit END
cases	::=	$\text{constset}_1 : \text{unit}_1, \cdots, \text{constset}_k : \text{unit}_k$
step	::=	LET $\text{letdecs}_1, \cdots, \text{letdecs}_k$.
	|	MAKE $\text{makedecs}_1, \cdots, \text{makedecs}_k$.
	|	JOIN $\text{joindecs}_1, \cdots, \text{joindecs}_k$.
letdecs	::=	signalname = unit
makedecs	::=	fnname : signalname
joindecs	::=	unit $\rightarrow$ signalname

Function Body

functionbody	::=	unit
	|	REFORM
	|	BIOP biopname
	|	DELAY (initial_value, ambig_time, ambig_value, delay_time)
	|	IDELAY (initial_value, delay_time)
	|	SAMPLE (interval, initial_value, skew_time)
	|	RAM (initial_value)

Type Declaration

typedec ::= TYPE typename = typeornew.

typeornew ::= type
| new

new ::= NEW tagname / (lwb .. upb)
| NEW (typealt_1 | $\cdots$ | typealt_k)
| NEW tagname ($'\text{char}_1$ | $\cdots$ | $'\text{char}_k$)

typealt ::= tagname & type
| altname

Function Declaration

fndec ::= FN fnname = (input) $\rightarrow$ type : functionbody.

input ::= type_1 : $\text{signalname}_1, \cdots, \text{type}_k$: signalname_k

Closure

closure ::= $\text{typedec}_1 \cdots \text{typedec}_j\ \text{fndec}_1 \cdots \text{fndec}_k$

Appendix D

Kernel of ELLA Data Structure

D.1 Basic Notation

Throughout this appendix the following notation will be used

abc	$\in$	Abc (ie. it is an element of the set Abc)
Indexer, Size, Fnno	$\subseteq$	$\mathbb{N}_1$
Typeno, Tagno, Inputno	$\subseteq$	$\mathbb{N}_1$
Signalno, Ambigtime, Delaytime	$\subseteq$	$\mathbb{N}_1$
Interval, Skew	$\subseteq$	$\mathbb{N}_1$
Inputtype, Outputtype	$\subseteq$	Type
Initialconst, Ambigconst	$\subseteq$	Const
Fnname, Biopname	$\subseteq$	”Upper case identifier or operator”
Name, Signalname	$\subseteq$	”Lower case identifier”
Typename, Tagname	$\subseteq$	”Lower case identifier”
Lowerbound, Upperbound	$\subseteq$	”positive or negative integer”
Character	$\subseteq$	”printable character”

D.2 Kernel Data Structure

Enumerated

Enumerated	::=	Enum
		| **string**(Typeno $\times$ TagnoSeq)
Enum	::=	**enum**(Typeno $\times$ Tagno)

Types

Type	::=	**typeno**(Typeno)
		| **typename**(Typename $\times$ Type)
		| **stringtype**(Size $\times$ Typeno)
		| **types**(TypeSeq)
Typety	::=	TypeOpt

Constants

Const ::= Enumerated
| **conststring**(Size × Const)
| **consts**(ConstSeq)
| **constassoc**(Enum × Const)
| **constquery**(Type)

Constant Sets

Constset ::= Enumerated
| **constsetalts**(ConstsetSeq)
| **constsetstring**(Size × Constset)
| **constsets**(ConstsetSeq)
| **constsetassoc**(Enum × Constset)
| **constsetany**(Type)

Units

Unit ::= Enumerated
| **conc**(Unit × Unit × Outputtype)
| **unitstring**(Size × Unit)
| **units**(UnitSeq)
| **call**(Fnno × Unit)
| **unitassoc**(Enum × Unit)
| **restrict**(Unit × Enum × Outputtype)
| **signal**(Signalno)
| **index**(Unit × Indexer × Outputtype)
| **unitquery**(Type)
| **caseclause**(Unit × CaseSeq × UnitOpt × Outputtype)

Case ::= **case**(Constset × Unit)

Function Declarations

Fndec ::= **fndec**(Fnname × Inputtype × SignaldecSeq × Outputtype × Fnbody)

Signaldec ::= **signaldec**(Signalname × Type × Unit | **input**)

Fnbody ::= Unit
| **reform**
| **biop**(Biopname)
| **delay**(Initialconst × Ambigtime × Ambigconst × Delaytime)
| **idelay**(Initialconst × Delaytime)
| **sample**(Interval × Initialconst × Skew)
| **ram**(Initialconst)

Type Declarations

Typedec	::=	**typedec**(Typename × New)
New	::=	**tags**(TagSeq)
	|	**ellaint**(Tagname × Lowerbound × Upperbound)
	|	**chars**(Tagname × CharacterSeq)
Tag	::=	**tag**(Tagname × TypeOpt)

Closures

Closure	::=	TypedecSeq × FndecSeq

Appendix E

FIFO Example

This appendix presents a description of a fifo written in high level ELLA together with an automatically transformed version of the circuit.

E.1 High Level FIFO

```
                          #  Type Declarations   #

TYPE bool = NEW ( t | f | x ),
     int  = NEW i/(0..100).


                # Fifo - data goes into the first empty cell. #
                # ----                                         #

MAC FIFO {INT size} = (int: data_in,  bool: shift_in shift_out) -> (int, bool):
(SEQ
   PVAR fifo ::= [size](i/0, f);                   # create state variable  #

   CASE shift_out
   OF  t : fifo := fifo[2..size] CONC (i/0, f)     # remove element         #
   ESAC;

   VAR entered := f;
   CASE shift_in                                   # add element            #
   OF  t : [INT i = 1..size-1]
              CASE (entered, fifo[i][2])
              OF  (f,f) : ( fifo[i] := (data_in, t);
                                entered := t
                          )
              ESAC
   ESAC;

   OUTPUT (fifo[1][1], entered)
```

```
).

FN FIFO_9 = (int: data_in,  bool: shift_in shift_out) -> (int, bool):
FIFO {9} (data_in, shift_in, shift_out).                        # call macro #
```

E.2 Transformed Description

This section presents the result of applying the built-in software transformations of ELLA to the above FIFO circuit. Note that names which begin with a '_' are automatically generated by the transformations. Also function names of the form 'NAME{..}' are instantiations of macros where the items between the braces are the parameters which have be used in the instantiation.

```
TYPE bool = NEW( t | f | x ).

TYPE int = NEW i/( 0..100 ).

FN _DELAY{ [ 9 ]( int, bool ), [ 9 ]( i/0, f ) } = ( [ 9 ]( int, bool )) ->
                                                    [ 9 ]( int, bool ):
   DELAY( [ 9 ]( i/0, f ), 1 ).

FN FIFO{ 9 } = ( int: data_in, bool: shift_in shift_out ) -> ( int, bool ):
   ( MAKE _DELAY{ [ 9 ]( int, bool ), [ 9 ]( i/0, f ) }: _s6fifo.
     LET _s7fifo =
         CASE shift_out
         OF t: ( LET _s8fifo =
                     _s6fifo[ 2..9 ] CONC ( i/0, f ).
                 OUTPUT _s8fifo
              )
         ELSE _s6fifo
         ESAC.
     LET _s9entered = f.
     LET ( fifo, entered ) =
          CASE shift_in
          OF t: ( LET ( _s12fifo, _s13entered ) =
                        CASE ( _s9entered,_s7fifo[1 ][ 2 ] )
                        OF ( f,f ):( LET _s14fifo = ( data_in,t ) CONC
                                                             _s7fifo[2..9 ].
                                     LET _s15entered = t.
                                     OUTPUT ( _s14fifo, _s15entered )
                                    )
                        ELSE ( _s7fifo, _s9entered )
                        ESAC.
                  LET ( _s16fifo, _s17entered ) =
                        CASE ( _s13entered, _s12fifo[2 ][ 2 ] )
                        OF ( f, f ): ( LET _s18fifo = ( _s12fifo[1 ] CONC
                                                      ( data_in, t )) CONC
                                                            _s12fifo[3..9 ].
```

```
                LET _s19entered = t.
                OUTPUT ( _s18fifo, _s19entered )
              )
      ELSE ( _s12fifo, _s13entered )
      ESAC.
LET ( _s20fifo, _s21entered ) =
      CASE ( _s17entered, _s16fifo[3 ][ 2 ] )
      OF ( f, f ): ( LET _s22fifo = ( _s16fifo[1..2 ] CONC
                                    ( data_in, t )) CONC
                                          _s16fifo[4..9 ].
                LET _s23entered = t.
                OUTPUT ( _s22fifo, _s23entered )
              )
      ELSE ( _s16fifo, _s17entered )
      ESAC.
LET ( _s24fifo, _s25entered ) =
      CASE ( _s21entered, _s20fifo[4 ][ 2 ] )
      OF ( f, f ): ( LET _s26fifo = ( _s20fifo[1..3 ] CONC
                                    ( data_in, t )) CONC
                                          _s20fifo[5..9 ].
                LET _s27entered = t.
                OUTPUT ( _s26fifo, _s27entered )
              )
      ELSE ( _s20fifo, _s21entered )
      ESAC.
LET ( _s28fifo, _s29entered ) =
      CASE ( _s25entered, _s24fifo[5 ][ 2 ] )
      OF ( f, f ): ( LET _s30fifo = ( _s24fifo[1..4 ] CONC
                                    ( data_in, t )) CONC
                                          _s24fifo[6..9 ].
                LET _s31entered = t.
                OUTPUT ( _s30fifo, _s31entered )
              )
      ELSE ( _s24fifo, _s25entered )
      ESAC.
LET ( _s32fifo, _s33entered ) =
      CASE ( _s29entered, _s28fifo[6 ][ 2 ] )
      OF ( f, f ): ( LET _s34fifo = ( _s28fifo[1..5 ] CONC
                                    ( data_in, t )) CONC
                                          _s28fifo[7..9 ].
                LET _s35entered = t.
                OUTPUT ( _s34fifo, _s35entered )
              )
      ELSE ( _s28fifo, _s29entered )
      ESAC.
LET ( _s36fifo, _s37entered ) =
      CASE ( _s33entered, _s32fifo[7 ][ 2 ] )
```

```
                        OF ( f, f ): ( LET _s38fifo = ( _s32fifo[1..6 ] CONC
                                                         ( data_in, t )) CONC
                                                               _s32fifo[8..9 ].
                                       LET _s39entered = t.
                                       OUTPUT ( _s38fifo, _s39entered )
                                     )
                        ELSE ( _s32fifo, _s33entered )
                        ESAC.
                  LET ( _s40fifo, _s41entered ) =
                        CASE ( _s37entered, _s36fifo[8 ][ 2 ] )
                        OF ( f, f ): ( LET _s42fifo = ( _s36fifo[1..7 ] CONC
                                                         ( data_in, t )) CONC
                                                                  _s36fifo[9 ].
                                       LET _s43entered = t.
                                       OUTPUT ( _s42fifo, _s43entered )
                                     )
                        ELSE ( _s36fifo, _s37entered )
                        ESAC.
                  OUTPUT ( _s40fifo, _s41entered )
                )
          ELSE ( _s7fifo, _s9entered )
          ESAC.
     JOIN fifo -> _s6fifo.
     OUTPUT ( fifo[ 1 ][ 1 ], entered )
   ).

FN FIFO_9 = ( int: data_in, bool: shift_in shift_out ) -> ( int, bool ):
   FIFO{ 9 }( data_in, shift_in, shift_out ).
```

It can be noted that if those names which begin with an underscore are replaced by valid ELLA names then the above description would be a description of the fifo in Core ELLA.

Appendix F

Three Pump Controller

F.1 Introduction

This appendix presents a high level, medium level and low level description in ELLA of a three pump controller. The definition of the controller is given in [Bar91] and is reproduced here

A reservoir is connected to a lake by a pipe line. Water is taken from the lake to the reservoir by a system of three pumps.
Three level sensors are installed on the reservoir. Their outputs are denoted by signals a_1, a_2, a_3. Signal a_i is 0 when the water is above level i, for i = 1,2,3 and has a value 1 when the water is below level i. The number of pumps that are on at any one time depends on the water level in the reservoir. in particular: if the water level is between level 1 and 2, then one pump should be in operation; if the water level is between level 2 and 3, then two pumps should be in operation; if the water level is below level 3, then three pumps should be in operation. Of course, if the water level is above level 1 then no pumps should be in operation. In order to equalise wear on the pumps, they should come into operation in a cyclic manner.

F.2 High Level ELLA Description

In this section we give a high level description of the pump controller. Three enumerated types have defined. The first 'pump' denotes which pumps are actually operating, the pumps being known as 'a', 'b' and 'c'. At first glance the ordering of the enumerated type might appear strange. However the ordering was chosen such that when the circuit is transformed to gate level the output of the controller will be a three bit signal with each bit representing one of the pumps. The second type 'level' denotes the level of water in the reservoir, with zero representing a full reservoir. The third type is a boolean flag which is used in the monitoring of the active pump. The function CONTROL is the pump controller and its CASE clause sets up which pumps get switched on.

```
TYPE  pump = NEW (none | a | b | ab | c | ca | bc | abc ),
      level = NEW l/(0..3),
      bool = NEW (t | f).

FN CONTROL = (level:in) -> pump:
( SEQ
      PVAR store ::= (none,t);
```

```
      store := CASE in OF
                 l/0 : (store[1],t),
                 l/1 : CASE store[1] OF
                             a | ca : (b,f),
                             b | ab : (c,f)
                             ELSE     (a,f)
                       ESAC,
                 l/2 : CASE store[1] OF
                             a | ab : (bc,f),
                             b | bc : (ca,f)
                             ELSE     (ab,f)
                       ESAC,
                 l/3 : (abc,f)
               ESAC;
      OUTPUT CASE store[2] OF
              t: none,
              f: store[1]
            ESAC
).
```

Although CONTROL has been written using sequences this is not really necessary. A functional version of CONTROL is therefore given, this being an equivalent description to the sequential form.

```
FN F1_DELAY = (( pump, bool )) -> ( pump, bool ): DELAY(( none, t ), 1 ).

FN CONTROL = ( level: in ) -> pump:
BEGIN
   MAKE F1_DELAY: s3store.
   LET store =
      CASE in OF
      l/0: ( s3store[ 1 ], t ),
      l/1:
         CASE s3store[ 1 ] OF
         a | ca: ( b, f ),
         b | ab: ( c, f )
         ELSE ( a, f )
         ESAC,
      l/2:
         CASE s3store[ 1 ] OF
         a | ab: ( bc, f ),
         b | bc: ( ca, f )
         ELSE ( ab, f )
         ESAC,
      l/3: ( abc, f )
      ESAC.
   JOIN store -> s3store.
   OUTPUT
      CASE store[ 2 ] OF
```

```
        t: none,
        f: store[ 1 ]
       ESAC
END.
```

F.3 Medium Level ELLA Description

This section presents the results of replacing the enumerated types for the pump switch's and level indicators by rows of two valued types. This synthesising of the types makes explicit the algorithm behind the type naming of the high level version. It would have been possible to describe the controller from the medium level from the outset, however the higher level version provides extra checks. In particular in the high level version the level indicators can only take four possible values whereas in this medium level version they can take eight. This medium level version treats such illegal values as 'unknown' and causes the simulator to return the ELLA unknown value, whereas the high level version would explicitly indicate if the level integer range was violated.

This medium level version has maintained close correspondence with the high level version by the use of 'constant' statements. Thus the majority of the controller description has remained unaltered, hence reducing the likelihood of error. The complete description is given by

```
TYPE  switch = NEW (on | off),
      pump   = [3]switch,
      level  = [3]switch,
      bool   = NEW (t | f).

CONST none = (off, off, off),
      a    = (on, off, off),
      b    = (off, on, off),
      c    = (off, off, on),
      ab   = (on, on, off),
      ca   = (on, off, on),
      bc   = (off, on, on),
      abc  = (on, on, on).

CONST level0 = (off, off, off),
      level1 = (on, off, off),
      level2 = (on, on, off),
      level3 = (on, on, on).


FN CONTROL = (level:in) -> pump:
( SEQ
      PVAR store ::= (none,t);
      store := CASE in OF
                  level0 : (store[1],t),
                  level1 : CASE store[1] OF
                              a | ca : (b,f),
                              b | ab : (c,f)
```

```
                         ELSE     (a,f)
                    ESAC,
              level2 : CASE store[1] OF
                         a | ab : (bc,f),
                         b | bc : (ca,f)
                         ELSE     (ab,f)
                    ESAC,
              level3 : (abc,f)
            ESAC;
      OUTPUT CASE store[2] OF
               t: none,
               f: store[1]
             ESAC
).
```

F.4 Low Level ELLA Description

This section presents the results of synthesising the medium level description of the pump controller through the ELLA-GATEMAP [Pit88] system. Apart from the functions F1_DELAY and CONTROL all the other functions are basic cells in one of the technology libraries that GATEMAP supports.

```
#-----------------------------------------------------------------------#
#                                                                       #
# ELLA netlist generated by GATEMAP II version 1.2                      #
#                                                                       #
# Module     : CONTROL                                                  #
# Date       : 16-MAY-1991 13:41                                        #
# Library    : USR$WORK:[]                                              #
# Technology : USR$GATEMAPROOT:[12.TECHNOLOGIES]*******                 #
#                                                                       #
#-----------------------------------------------------------------------#

TYPE bool = NEW( f | t | x | z ).

TYPE tech_bool = bool.

CONST logic_0 = f.

#-------------------------- LIBRARY CELLS -----------------------------#

FN INV1 = ( bool: a ) -> bool: # Inverter #.

FN NAND2 = ( bool: a b ) -> bool: # Two Input NAND #

FN NAND3 = ( bool: a b c ) -> bool: # Three Input NAND #
```

```
FN NOR2 = ( bool: a b ) -> bool: # Two Input NOR #

FN NOR3 = ( bool: a b c ) -> bool: # Three Input NOR #

FN X2ANOR = ( bool: a b c d ) -> bool: NOR(AND(a,b), AND(c,d)).

FN EXNOR = ( bool: a b ) -> bool: # Two Input Exclusive OR #

FN CLKB = ( bool: ai ) -> ( bool, bool ): # Clock Drvier #

FN DF = ( bool: ckt cki d ) -> ( bool, bool ):  # Clocked Cell #

#---------------------- ELLA DELAY FUNCTION --------------------------#

FN F1_DELAY = ( tech_bool: unnamed_input_1, tech_bool: unnamed_input_2,
                tech_bool: unnamed_input_3, tech_bool: unnamed_input_4 ) ->
              ( tech_bool, tech_bool, tech_bool, tech_bool ):
   BEGIN
       MAKE DF  : xcmp17 xcmp19 xcmp15 xcmp21,
            CLKB: xcmp18.
       JOIN ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_3 ) -> xcmp17,
            ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_2 ) -> xcmp19,
            ( logic_0 ) -> xcmp18,
            ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_1 ) -> xcmp15,
            ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_4 ) -> xcmp21.
       OUTPUT ( xcmp15[ 1 ], xcmp19[ 1 ], xcmp17[ 1 ], xcmp21[ 1 ] )
   END.

#------------------------- PUMP CONTROLLER ----------------------------#

FN CONTROL = ( tech_bool: in_1, tech_bool: in_2, tech_bool: in_3 ) ->
             ( tech_bool, tech_bool, tech_bool ):
   BEGIN
       MAKE INV1    : xcmp39 xcmp69 xcmp76 xcmp24 xcmp74 xcmp33 xcmp61 xcmp37,
            NAND2   : xcmp56 xcmp66 xcmp25 xcmp36 xcmp38 xcmp53 xcmp70
                      xcmp47 xcmp64,
            NAND3   : xcmp45 xcmp67 xcmp84 xcmp35,
            NOR3    : xcmp75 xcmp40 xcmp80,
            X2ANOR  : xcmp78 xcmp72 xcmp82,
            EXNOR   : xcmp48,
            F1_DELAY: xcmp4.
       JOIN ( xcmp72, xcmp67 ) -> xcmp56,
            ( xcmp37, in_2, xcmp47 ) -> xcmp45,
            ( xcmp56 ) -> xcmp39,
            ( xcmp76, xcmp4[ 1 ], xcmp74 ) -> xcmp75,
            ( xcmp37, in_2, xcmp66 ) -> xcmp35,
            ( in_1, xcmp4[ 2 ], xcmp37, xcmp80 ) -> xcmp78,
```

```
        ( xcmp61, xcmp4[ 2 ] ) -> xcmp66,
        ( in_1, xcmp4[ 1 ], xcmp37, xcmp75 ) -> xcmp72,
        ( xcmp53 ) -> xcmp69,
        ( in_2, xcmp4[ 1 ], xcmp4[ 3 ] ) -> xcmp40,
        ( in_3 ) -> xcmp76,
        ( xcmp24, xcmp37 ) -> xcmp25,
        ( in_1, xcmp4[ 3 ], in_3, xcmp48 ) -> xcmp82,
        ( xcmp78, xcmp35 ) -> xcmp36,
        ( xcmp37, in_2, xcmp64 ) -> xcmp67,
        ( xcmp39, xcmp37 ) -> xcmp38,
        ( xcmp37, xcmp4[ 1 ], xcmp4[ 2 ] ) -> xcmp84,
        ( xcmp36 ) -> xcmp24,
        ( xcmp4[ 3 ] ) -> xcmp74,
        ( xcmp4[ 2 ] ) -> xcmp33,
        ( xcmp4[ 1 ] ) -> xcmp61,
        ( xcmp82, xcmp45 ) -> xcmp53,
        ( in_1 ) -> xcmp37,
        ( xcmp69, xcmp37 ) -> xcmp70,
        ( xcmp33, xcmp4[ 3 ] ) -> xcmp47,
        ( xcmp66, xcmp47 ) -> xcmp64,
        ( xcmp76, xcmp61, xcmp4[ 2 ] ) -> xcmp80,
        ( xcmp40, xcmp84 ) -> xcmp48,
        ( xcmp56, xcmp36, xcmp53, xcmp37 ) -> xcmp4.
    OUTPUT ( xcmp38, xcmp25, xcmp70 )
END.
```

Appendix G

Built-in-Operators of the Kernel of ELLA

G.1 Introduction

In this appendix we give the Semantics of Machine Independent Built-In Operators (BIOPs) for the **Kernel**. The complete definition of all available BIOPs can be found in [Tai88a]. Within the **Kernel** an enumerated value is referred to by tuples. For enumerated values that apply to a BIOP the first item is an integer which represents the type number. The second is either an integer representing the enumerated value that the signal takes, or a structure of integers representing values of an ELLA string. The following terminology will be used in this section

USBS_n	=	Unsigned bit string of length 'n'
SBS_n	=	Signed bit string of length 'n'
$\mathbf{bit}_m^v$	=	a bit string of size 'm' with elements set to 'v'

It should be noted that conversions from the **Kernel**'s representation of strings both to and from USBS's and SBS's are needed before some of the BIOPs can be called from the **Kernel**. In the next subsection we define some basic functions which are needed for the definition of the BIOPs

G.2 Basic Functions

This subsection defines three sets of functions, first the definition of functions for converting to/from two valued enumerated types and boolean values. Second, functions for mapping bit strings into integers and lastly the quotient result for integers.

G.2.1 Enumerated

Define **n2b** such that:
n2b: $\mathbb{N} \longrightarrow$ BOOL
n2b(v) $\underline{\triangle}$ IF v = 1 THEN **false** ELSE **true**

Define **b2n** such that:
b2n: BOOL $\longrightarrow \mathbb{N}$
b2nv $\underline{\triangle}$ IF v THEN 2 ELSE 1

G.2.2 Strings

Define 'unsigned_string', **ust**, which evaluates the integer value of an unsigned bit strings as:
ust: $USBS_n \longrightarrow \mathbb{N}$
ust(v) $\underline{\triangle}$ Integer value corresponding to v

Define 'unsigned_string_inverse', $\mathbf{ust}_n^{-1}$ as:
$\mathbf{ust}_n^{-1}$: $\mathbb{N} \longrightarrow USBS_n$
$\mathbf{ust}_n^{-1}$(v) $\underline{\triangle}$ Unsigned bit string representation of v
Define 'signed_string', **sst**, which evaluates the integer value of a signed bit string as:
sst: $SBS_n \longrightarrow \mathbb{N}$
sst(v) $\underline{\triangle}$ Integer value corresponding to v

Define 'signed_string_inverse', $\mathbf{sst}^{-1}$ as:
$\mathbf{sst}_n^{-1}$: $\mathbb{N} \longrightarrow SBS_n$
$\mathbf{sst}_n^{-1}$(v) $\underline{\triangle}$ Signed bit string representation of v

G.2.3 Quotient

Define 'OVER' such that:
OVER : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
v_1 OVER v_2 $\underline{\triangle}$ Quotient of v_1 over v_2
The following sections give the definition of the BIOPs available in the **Kernel**.

G.3 Built In Logical Operators

G.3.1 Two Valued Enumerated Types

not: $\mathbb{N} \longrightarrow \mathbb{N}$
not $\underline{\triangle}$ λv . **b2n**($\neg$ **n2b**(v))

and: $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
and $\underline{\triangle}$ $\lambda(v_1, v_2)$. **b2n**(**n2b**(v_1) $\wedge$ **n2b**(v_2))

or: $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
or $\underline{\triangle}$ $\lambda(v_1, v_2)$. **b2n**(**n2b**(v_1) $\vee$ **n2b**(v_2))

xor: $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
xor $\underline{\triangle}$ $\lambda(v_1, v_2)$. **b2n**(**n2b**(v_1) $\oplus$ **n2b**(v_2))

G.3.2 Bit Strings

and: $USBS_n \times USBS_n \longrightarrow USBS_n$
and $\underline{\triangle}$ $\lambda(v_1, v_2)$. ([INT i=1..n] v_1[i] $\wedge$ v_2[i])

$\text{or: USBS}_n \times \text{USBS}_n \longrightarrow \text{USBS}_n$
$\text{or} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ ([\text{INT } i{=}1..n]\ v_1[i] \vee v_2[i])$

$\text{xor: USBS}_n \times \text{USBS}_n \longrightarrow \text{USBS}_n$
$\text{xor} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ ([\text{INT } i{=}1..n]\ v_1[i] \oplus v_2[i])$

$\text{not: USBS}_n \longrightarrow \text{USBS}_n$
$\text{not} \underline{\triangle}\ \lambda v\ .\ ([\text{INT } i{=}1..n]\ \neg\ v[i])$

G.4 Built In Relational Operations

G.4.1 Enumerated Types

$\text{eq: } \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
$\text{eq} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ \mathbf{b2n}(v_1 = v_2)$

$\text{gt: } \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
$\text{gt} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ \mathbf{b2n}(v_1 > v_2)$

$\text{ge: } \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
$\text{ge} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ \mathbf{b2n}(v_1 \geq v_2)$

$\text{lt: } \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
$\text{lt} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ \mathbf{b2n}(v_1 < v_2)$

$\text{le: } \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
$\text{le} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ \mathbf{b2n}(v_1 \leq v_2)$

G.4.2 Unsigned Bit-Strings

$\text{eq\_us: USBS}_n \times \text{USBS}_m \longrightarrow \mathbb{N}$
$\text{eq\_us} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ \text{eq}(\ \mathbf{ust} v_1,\ \mathbf{ust} v_2)$

$\text{gt\_us: USBS}_n \times \text{USBS}_m \longrightarrow \mathbb{N}$
$\text{gt\_us} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ \text{gt}(\ \mathbf{ust} v_1,\ \mathbf{ust} v_2)$

$\text{ge\_us: USBS}_n \times \text{USBS}_m \longrightarrow \mathbb{N}$
$\text{ge\_us} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ \text{ge}(\ \mathbf{ust} v_1,\ \mathbf{ust} v_2)$

$\text{lt\_us: USBS}_n \times \text{USBS}_m \longrightarrow \mathbb{N}$
$\text{lt\_us} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ \text{lt}(\ \mathbf{ust} v_1,\ \mathbf{ust} v_2)$

$\text{le\_us: USBS}_n \times \text{USBS}_m \longrightarrow \mathbb{N}$
$\text{le\_us} \underline{\triangle}\ \lambda(v_1, v_2)\ .\ \text{le}(\ \mathbf{ust} v_1,\ \mathbf{ust} v_2)$

G.4.3 Signed Bit-Strings

eq_s: $\mathrm{SBS}_n \times \mathrm{SBS}_m \longrightarrow \mathbb{N}$
eq_s $\underline{\triangle}$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. eq($\mathbf{sst}\mathrm{v}_1$, $\mathbf{sst}\mathrm{v}_2$)

gt_s: $\mathrm{SBS}_n \times \mathrm{SBS}_m \longrightarrow \mathbb{N}$
gt_s $\underline{\triangle}$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. gt($\mathbf{sst}\mathrm{v}_1$, $\mathbf{sst}\mathrm{v}_2$)

ge_s: $\mathrm{SBS}_n \times \mathrm{SBS}_m \longrightarrow \mathbb{N}$
ge_s $\underline{\triangle}$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. ge($\mathbf{sst}\mathrm{v}_1$, $\mathbf{sst}\mathrm{v}_2$)

lt_s: $\mathrm{SBS}_n \times \mathrm{SBS}_m \longrightarrow \mathbb{N}$
lt_s $\underline{\triangle}$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. lt($\mathbf{sst}\mathrm{v}_1$, $\mathbf{sst}\mathrm{v}_2$)

le_s: $\mathrm{SBS}_n \times \mathrm{SBS}_m \longrightarrow \mathbb{N}$
le_s $\underline{\triangle}$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. le($\mathbf{sst}\mathrm{v}_1$, $\mathbf{sst}\mathrm{v}_2$)

G.5 Built In Shift Operations

sl-us: $\mathbb{N} \longrightarrow \mathrm{USBS}_n \longrightarrow \mathrm{USBS}_{n+m}$
sl-us $\underline{\triangle}$ λm . λv . (v $\frown$ $\mathbf{bit}^0_m$)

sl-s: $\mathbb{N} \longrightarrow \mathrm{SBS}_n \longrightarrow \mathrm{SBS}_{n+m}$
sl-s $\underline{\triangle}$ λm . λv . (v $\frown$ $\mathbf{bit}^0_m$)

sr-us: $\mathbb{N} \longrightarrow \mathrm{USBS}_n \longrightarrow \mathrm{USBS}_{n+m}$
sr-us $\underline{\triangle}$ λm . λv . ($\mathbf{bit}^0_m$ $\frown$ v)

sr-s: $\mathbb{N} \longrightarrow \mathrm{SBS}_n \longrightarrow \mathrm{SBS}_{n+m}$
sr-s $\underline{\triangle}$ λm . λv . ($\mathbf{bit}^{v[1]}_m$ $\frown$ v)

G.6 Built In Arithmetic Operations

G.6.1 Unsigned Bit Strings

plus_us: $\mathrm{USBS}_n \times \mathrm{USBS}_m \longrightarrow \mathrm{USBS}_{MAX\{m,n\}+1}$
plus_us $\underline{\triangle}$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. $\mathbf{ust}^{-1}_{MAX\{m,n\}+1}$ ($\mathbf{ust}\mathrm{v}_1$ + $\mathbf{ust}\mathrm{v}_2$)

minus_us: $\mathrm{USBS}_n \times \mathrm{USBS}_m \longrightarrow \mathrm{USBS}_{MAX\{m,n\}+1}$
minus_us $\underline{\triangle}$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. $\mathbf{ust}^{-1}_{MAX\{m,n\}+1}$ ($\mathbf{ust}\mathrm{v}_1$ - $\mathbf{ust}\mathrm{v}_2$)

negate_us: $\mathrm{USBS}_n \longrightarrow \mathrm{USBS}_{n+1}$
negate_us $\underline{\triangle}$ λv . $\mathbf{ust}^{-1}_{n+1}(\neg$ $\mathbf{ust}$v)

times_us: $\mathrm{USBS}_n \times \mathrm{USBS}_m \longrightarrow \mathrm{USBS}_{m+n}$
times_us $\triangleq$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. $\mathbf{ust}^{-1}_{m+n}\ (\mathrm{v}_1 * \mathrm{v}_2)$

divide_us: $\mathrm{USBS}_n \times \mathrm{USBS}_m \longrightarrow \mathbb{N} \times \mathrm{USBS}_n \times \mathrm{USBS}_m$
divide_us $\triangleq$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. ($\mathbf{b2n}(\ \mathbf{ust}\mathrm{v}_2 = 0)$, $\mathbf{ust}^{-1}_n(\ \mathbf{ust}\mathrm{v}_1\ \mathrm{OVER}\ \mathbf{ust}\mathrm{v}_2)$,
$\mathbf{ust}^{-1}_m(\ \mathbf{ust}\mathrm{v}_1 - ((\ \mathbf{ust}\mathrm{v}_1\ \mathrm{OVER}\ \mathbf{ust}\mathrm{v}_2) * \mathbf{ust}\mathrm{v}_2)\ )$

sqrt_us: $\mathrm{USBS}_n \longrightarrow \mathrm{USBS}_{(n+1)\%2}$
sqrt_us $\triangleq$ $\lambda\mathrm{v}$. $\mathbf{ust}^{-1}_{(n+1)\%2}(\sqrt{\ }\ \mathbf{ust}\mathrm{v})$

mod_us: $\mathrm{USBS}_n \times \mathrm{USBS}_m \longrightarrow \mathrm{USBS}_m$
mod_us $\triangleq$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. $\mathbf{ust}^{-1}_m\ (\ \mathbf{ust}\mathrm{v}_1\ \mathrm{MOD}\ \mathbf{ust}\mathrm{v}_2)$

range_us: $\mathbb{N} \longrightarrow \mathrm{USBS}_n \longrightarrow \mathbb{N} \times \mathrm{USBS}_m$
range_us $\triangleq$ $\lambda\mathrm{m}$. $\lambda\mathrm{v}$. ($\mathbf{b2n}(\ \mathbf{ust}\mathrm{v} \geq 2^m)$, $\mathbf{ust}^{-1}_m(\ \mathbf{ust}\mathrm{v})$)

G.6.2 Signed Bit Strings

plus_s: $\mathrm{SBS}_n \times \mathrm{SBS}_m \longrightarrow \mathrm{SBS}_{MAX\{m,n\}+1}$
plus_s $\triangleq$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. $\mathbf{sst}^{-1}_{MAX\{m,n\}+1}\ (\ \mathbf{sst}\mathrm{v}_1 + \mathbf{sst}\mathrm{v}_2)$

minus_s: $\mathrm{SBS}_n \times \mathrm{SBS}_m \longrightarrow \mathrm{SBS}_{MAX\{m,n\}+1}$
minus_s $\triangleq$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. $\mathbf{sst}^{-1}_{MAX\{m,n\}+1}\ (\ \mathbf{sst}\mathrm{v}_1 - \mathbf{sst}\mathrm{v}_2)$

negate_s: $\mathrm{SBS}_n \longrightarrow \mathrm{SBS}_{n+1}$
negate_s $\triangleq$ $\lambda\mathrm{v}$. $\mathbf{sst}^{-1}_{n+1}(\neg\ \mathbf{sst}\mathrm{v})$

times_s: $\mathrm{SBS}_n \times \mathrm{SBS}_m \longrightarrow \mathrm{SBS}_{m+n}$
times_s $\triangleq$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. $\mathbf{sst}^{-1}_{m+n}\ (\mathrm{v}_1 * \mathrm{v}_2)$

divide_s: $\mathrm{SBS}_n \times \mathrm{SBS}_m \longrightarrow \mathbb{N} \times \mathrm{SBS}_n \times \mathrm{SBS}_m$
divide_s $\triangleq$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. ($\mathbf{b2n}(\ \mathbf{sst}\mathrm{v}_2 = 0)$, $\mathbf{sst}^{-1}_n(\ \mathbf{sst}\mathrm{v}_1\ \mathrm{OVER}\ \mathbf{sst}\mathrm{v}_2)$,
$\mathbf{sst}^{-1}_m(\ \mathbf{sst}\mathrm{v}_1 - ((\ \mathbf{sst}\mathrm{v}_1\ \mathrm{OVER}\ \mathbf{sst}\mathrm{v}_2) * \mathbf{sst}\mathrm{v}_2)\ )$

mod_s: $\mathrm{SBS}_n \times \mathrm{SBS}_m \longrightarrow \mathrm{SBS}_m$
mod_s $\triangleq$ $\lambda(\mathrm{v}_1, \mathrm{v}_2)$. $\mathbf{sst}^{-1}_m\ (\ \mathbf{sst}\mathrm{v}_1\ \mathrm{MOD}\ \mathbf{sst}\mathrm{v}_2)$

range_s: $\mathbb{N} \longrightarrow \mathrm{SBS}_n \longrightarrow \mathbb{N} \times \mathrm{SBS}_m$
range_s $\triangleq$ $\lambda\mathrm{m}$. $\lambda\mathrm{v}$. ($\mathbf{b2n}(\ \mathbf{sst}\mathrm{v} \geq 2^{m-1} \vee \mathbf{sst}\mathrm{v} < -2^{m-1})$, $\mathbf{sst}^{-1}_m(\ \mathbf{sst}\mathrm{v})$)

abs_s: $\mathrm{SBS}_n \longrightarrow \mathrm{SBS}_n$
abs_s $\triangleq$ $\lambda\mathrm{v}$. $\mathbf{sst}^{-1}_n(|\ \mathbf{sst}\mathrm{v}\ |\ )$

G.7 Built In Transformation Operations

transform_us: $\mathbb{N} \longrightarrow \mathbb{N} \longrightarrow \mathbb{N} \times USBS_m$
transform_us $\underline{\triangle}$ λm . λv . (**b2n**(v $<$ 2^m), $\mathbf{ust}_m^{-1}$v)

transform_us: $\mathbb{N} \longrightarrow USBS_n \longrightarrow \mathbb{N} \times \mathbb{N}$
transform_us $\underline{\triangle}$ λm . λv . (**b2n**(**ust**v $<$ m), **ust**v)

transform_s: $\mathbb{N} \longrightarrow \mathbb{N} \longrightarrow \mathbb{N} \times SBS_m$
transform_s $\underline{\triangle}$ λm . λv . (**b2n**($-2^{m-1} \leq$ v $< 2^{m-1}$), $\mathbf{sst}_m^{-1}$v)

transform_s: $\mathbb{N} \longrightarrow SBS_n \longrightarrow \mathbb{N} \times \mathbb{N}$
transform_s $\underline{\triangle}$ λm . λv . (**b2n**(**sst**v $<$ m), **sst**v)

Appendix H

BKE Semantic definitions

H.1 IO Automata and Hardware Components

$$
\begin{aligned}
Trans :: \; & from : State \\
& ivals : Name \xrightarrow{m} Val \\
& ovals : Name \xrightarrow{m} Val \\
& to : State
\end{aligned}
$$

$$
\begin{aligned}
IOA :: \; & states : State\text{-}\mathbf{set} \\
& inputs : Name\text{-}\mathbf{set} \\
& outputs : Name\text{-}\mathbf{set} \\
& transitions : Trans\text{-}\mathbf{set} \\
& start : State\text{-}\mathbf{set}
\end{aligned}
$$

$$inv\text{-}IOA : state\text{-}\mathbf{set} \times Name\text{-}\mathbf{set} \times Name\text{-}\mathbf{set} \times Trans\text{-}\mathbf{set} \times State\text{-}\mathbf{set} \to \mathbb{B}$$

$$
\begin{aligned}
& inv\text{-}IOA(ss, i, o, ts, st) \quad \triangleq \quad st \subseteq ss \wedge i \cap o = \{\,\} \wedge \forall t \in ts \cdot \\
& \quad from(i) \in ss \wedge to(i) \in ss \wedge \mathbf{dom}\, ivals(t) = i \wedge \mathbf{dom}\, ovals(t) = o
\end{aligned}
$$

H.1.1 Product IO Automata

$$causal\text{-}prod : IOA \times IOA \to IOA$$

$$
\begin{aligned}
& causal\text{-}prod(a_1, a_2) \quad \triangleq \\
& \quad \mathbf{let}\; mk\text{-}IOA(ss_1, i_1, o_1, ts_1, st_1) = a_1 \;\; \mathbf{in} \\
& \quad \mathbf{let}\; mk\text{-}IOA(ss_2, i_2, o_2, ts_2, st_2) = a_2 \;\; \mathbf{in} \\
& \quad \mathbf{let}\; o = o_1 \cup o_2 \;\; \mathbf{in} \\
& \quad \mathbf{let}\; i = (i_1 \cup i_2) - o \;\; \mathbf{in} \\
& \quad \mathbf{let}\; ts = \{mk\text{-}Trans((s_1, s_2), im, om, (s_1', s_2')) \mid \\
& \qquad\qquad \exists (s_1, im_1, om_1, s_1') \in ts_1, (s_2, im_2, om_2, s_2') \in ts_2 \cdot \\
& \qquad\qquad \mathbf{let}\; vals = im_1 \cup im_2 \cup om_1 \cup om_2 \;\; \mathbf{in} \\
& \qquad\qquad im = i \lhd vals \wedge om = o \lhd vals\} \\
& \qquad \mathbf{in} \\
& \quad mk\text{-}IOA(ss_1 \times ss_2, i, o, new\text{-}ts, st_1 \times st_2)
\end{aligned}
$$

$$\mathbf{pre}\; outputs(a_1) \cap outputs(a_2) = \{\,\}$$

H.1.2 Semantic Objects

$Name = Id \cup \mathbb{Z} \setminus \{0\}$

$SemObj = IOA$

where

$inv\text{-}SemObj : IOA \to \mathbb{B}$

$inv\text{-}SemObj(mk\text{-}IOA(ss, i, o, ts, st)) \quad \triangleq \quad \exists n \in \mathbb{N} \cdot o = \{1, \ldots, n\}$

H.1.3 Expressions

$Expression = Const \cup Var \cup Let \cup Row \cup Case \cup Funapp \cup Delays \cup Sample \cup Not \cup Infix$

$Env = Name \xrightarrow{m} SemObj$

$MExpression : Expression \to Env \to SemObj$

$$
\begin{array}{lll}
MExpression[\![e]\!]\sigma & \triangleq & \textbf{cases } e \textbf{ of} \\
 & & \begin{array}{ll}
 mk\text{-}Const(v) & \to MConst[\![e]\!] \\
 mk\text{-}Var(v) & \to MVar[\![e]\!] \\
 mk\text{-}Let(bd, e_1) & \to MLet[\![e]\!]\sigma \\
 mk\text{-}Row(els) & \to MRow[\![els]\!]\sigma \\
 mk\text{-}Case(rw, itms, else) & \to MCase[\![e]\!]\sigma \\
 mk\text{-}Funapp(f, args) & \to MFunapp[\![e]\!]\sigma \\
 mk\text{-}IDelay(e_1, v, n) & \to MIDelay[\![e]\!]\sigma \\
 mk\text{-}Delay(e_1, v_1, n_1, v_2, n_2) & \to MDelay[\![e]\!]\sigma \\
 mk\text{-}Sample(e_1, v, n_1, n_2) & \to MSample[\![e]\!]\sigma \\
 mk\text{-}Not(e_1) & \to MNot[\![e]\!]\sigma \\
 mk\text{-}Infix(e_1, op, e_2) & \to MInfix[\![e]\!]\sigma
 \end{array} \\
 & & \textbf{end}
\end{array}
$$

H.1.3.1 Constants

$Const \;::\; Val$

$MConst : Const \to SemObj$

$MConst[\![mk\text{-}Const(c)]\!] \quad \triangleq \quad const\text{-}machine(c)$

$const\text{-}machine : Val \to SemObj$

$const\text{-}machine(c) \quad \triangleq \quad mk\text{-}IOA(\{0\}, \{\,\}, \{1\}, \{mk\text{-}Trans(0, \{\,\}, \{1 \mapsto c\}, 0)\}, \{0\})$

H.1.3.2 Variables

$Var :: Id$

$MVar : Var \rightarrow SemObj$

$MVar[\![mk\text{-}Var(x)]\!] \quad \triangleq \quad var\text{-}machine(x)$

$var\text{-}machine : Id \rightarrow SemObj$

$$
\begin{array}{l}
var\text{-}machine(x) \quad \triangleq \\
\quad \textbf{let } ts = \{mk\text{-}Trans(0, \{x \mapsto v\}, \{1 \mapsto v\}, 0) \mid v \in Val\} \ \textbf{ in} \\
\quad mk\text{-}IOA(\{0\}, \{x\}, \{1\}, ts, \{0\})
\end{array}
$$

H.1.3.3 Recursive Let

$$
\begin{array}{lrcl}
Let :: & eqns & : & Binding \\
 & return & : & Expression
\end{array}
$$

$MLet : Let \rightarrow Env \rightarrow SemObj$

$$
\begin{array}{l}
MLet[\![mk\text{-}Let(eqs, ret)]\!]\sigma \quad \triangleq \\
\quad \textbf{let } expr\text{-}obj = MExpression[\![ret]\!]\sigma \ \textbf{ in} \\
\quad \textbf{let } outs = outputs(expr\text{-}obj) \ \textbf{ in} \\
\quad restrict(causal\text{-}product(Mbinding[\![eqns]\!]\sigma, expr\text{-}obj), outs)
\end{array}
$$

$$
\begin{array}{lrcl}
BindElem :: & lhs & : & Name^{+} \\
 & rhs & : & Expression \cup SemObj
\end{array}
$$

$BindList = BindElem^{+}$

$inv\text{-}BindList : BindElem^{+} \rightarrow \mathbb{B}$

$inv\text{-}BindList(bl) \quad \triangleq \quad \forall i, j \cdot i \neq j \ \Rightarrow \ (\textbf{elems}\, lhs(bl(i)) \cap \textbf{elems}\, lhs(bl(j))) = \{\,\}$

$Binding :: BindList$

$inv\text{-}Binding : BindList \rightarrow \mathbb{B}$

$inv\text{-}Binding(mk\text{-}Binding(l)) \quad \triangleq \quad \forall x \in \textbf{elems}\, l \cdot rhs(x) \in Expression$

$MBinding : Binding \rightarrow Env \rightarrow IOA$

$MBinding[\![mk\text{-}Binding(bdl)]\!]\sigma \quad \triangleq \quad bind(eval\text{-}rhs(bdl, \sigma))$

$eval\text{-}rhs : BindList \times Env \to BindList$

$$
\begin{aligned}
& eval\text{-}rhs(bdl, \sigma) \quad \triangleq \\
& \quad \textbf{cases } bdl \textbf{ of} \\
& \quad [mk\text{-}BindElem(x,e)] \quad \to [mk\text{-}BindElem(x, MExpression[\![e]\!]\sigma)] \\
& \quad [mk\text{-}BindElem(x,e)] \frown l \to mk\text{-}BindElem(x, MExpression[\![e]\!]\sigma) \frown eval\text{-}rhs(l) \\
& \quad \textbf{end}
\end{aligned}
$$

$\textbf{pre } \forall x \in \textbf{elems}\, bdl \cdot rhs(x) \in Expression$

$EvBindList = BindList$

where

$inv\text{-}EvBindList : BindElem^{+} \to \mathbb{B}$

$inv\text{-}EvBindList(l) \quad \triangleq \quad \forall x \in \textbf{elems}\, l \cdot rhs(x) \in SemObj$

$bind : EvBindList \to IOA$

$$
\begin{aligned}
bind(bdlist) \quad \triangleq \quad & \textbf{cases } bdlist \textbf{ of} \\
& [mk\text{-}BindElem(x,a)] \quad \to rename(a,x) \\
& l \frown [mk\text{-}BindElem(x,a)] \to causal\text{-}prod(bind(l), rename(a,x)) \\
& \textbf{end}
\end{aligned}
$$

$rename : SemObj \times Name^{+} \to IOA$

$$
\begin{aligned}
& rename(a, xl) \quad \triangleq \\
& \quad \textbf{let } mk\text{-}IOA(ss, i, o, ts, s) = a \textbf{ in} \\
& \quad \textbf{let } inds = \{ j \mid xl(j) \in i \} \textbf{ in} \\
& \quad \textbf{let } new\text{-}ts = \{ rename\text{-}trans(t, xl) \mid t \in ts \land \forall j \in inds \cdot ivals(t)(xl(j)) = ovals(t)(j) \} \textbf{ in} \\
& \quad mk\text{-}IOA(ss, i - \textbf{elems}\, xl, \textbf{elems}\, xl, new\text{-}ts, s)
\end{aligned}
$$

$\textbf{pre } outputs(a) = \textbf{inds}\, xl \land \textbf{card}\, (\textbf{elems}\, xl) = \textbf{len}\, xl$

$rename\text{-}trans : Trans \times Name^{+} \to IOA$

$$
\begin{aligned}
& rename\text{-}trans(t, xl) \quad \triangleq \\
& \quad \textbf{let } mk\text{-}Trans(fr, iv, ov, to) = t \textbf{ in} \\
& \quad mk\text{-}Trans(fr, \textbf{elems}\, x \lhd iv, \{ xl(i) \mapsto ov(i) \mid i \in \textbf{inds}\, xl \}, to)
\end{aligned}
$$

$\textbf{pre dom}\, ovals(t) = \textbf{inds}\, xl$

H.1.3.4 Rows

$Row = Expression^{+}$

$MRow : Row \to Env \to SemObj$

$MRow[\![els]\!]\sigma \quad \triangleq \quad MBinding(number\text{-}outputs(els))$

$number\text{-}outputs : Expression^{+} \rightarrow Binding$

$number\text{-}outputs(els) \quad \triangleq$
  cases els **of**
  $[\,] \qquad\qquad \rightarrow [\,]$
  $el \frown [expr] \rightarrow [mk\text{-}BindElem([\mathbf{len}\, els], expr)] \frown number\text{-}outputs(el)$
  end

H.1.3.5 Case expressions

$ChoiceVal = Val \cup \{\mathrm{X}\}$

$RowChoiceVal = ChoiceVal^{+}$

$$\begin{array}{lrcl} Case & :: \; selector & : & Row \\ & items & : & CaseItem^{+} \\ & else\text{-}row & : & Row \end{array}$$

$$\begin{array}{lrcl} CaseItem & :: \; chooser & : & RowChoiceVal \\ & result & : & Row \end{array}$$

$MCase : Case \rightarrow Env \rightarrow SemObj$

$MCase[\![mk\text{-}Case(selector, itms, else\text{-}row)]\!]\sigma \quad \triangleq$
  let $k = \mathbf{len}\, selector$ **in**
  let $n = \mathbf{len}\, itms$ **in**
  let $s = \mathbf{len}\, result(\mathbf{hd}\, itms)$ **in**
  let $selector\text{-}ioa = MRow[\![selector]\!]\sigma$ **in**
  let $chooser\text{-}ioa = MChList(get\text{-}choosers(itms))$ **in**
  let $choices\text{-}ioa = iter\text{-}causal\text{-}prod(get\text{-}choices\text{-}rename(k, s, itms, \sigma))$ **in**
  let $else\text{-}ioa = rename(MExpression[\![else\text{-}row]\!]\sigma, outseq(k, n, s))$ **in**
  $iter\text{-}causal\text{-}prod([selector\text{-}ioa, chooser\text{-}ioa, choices\text{-}ioa, else\text{-}ioa])$

$$
\begin{array}{l}
\mathit{MChList} : \mathit{RowChoiceVal}^{+} \to \mathit{IOA}\\[1ex]
\mathit{MChList}(\mathit{rch\text{-}list}) \quad \triangleq\\
\quad \textbf{let } k = \textbf{len}\, \mathit{chooser}(\textbf{hd}\, \mathit{rch\text{-}list}) \;\; \textbf{in}\\
\quad \textbf{let } n = \textbf{len}\, \mathit{rch\text{-}list} \;\; \textbf{in}\\
\quad \textbf{let } s = \textbf{len}\, \mathit{result}(\textbf{hd}\, \mathit{rch\text{-}list}) \;\; \textbf{in}\\
\quad \textbf{let } \mathit{input\text{-}set} = \{-(k+(n+1)s), \ldots, -1\} \;\; \textbf{in}\\
\quad \textbf{let } \mathit{output\text{-}set} = \{1, \ldots, s\} \;\; \textbf{in}\\
\quad \textbf{let } \mathit{poss\text{-}alt\text{-}maps} = \{-s, \ldots, -1\} \xrightarrow{m} \mathit{Val} \;\; \textbf{in}\\
\quad \textbf{let } \mathit{poss\text{-}chooser\text{-}maps} = \{-k, \ldots, -1\} \xrightarrow{m} \mathit{Val} \;\; \textbf{in}\\
\quad \textbf{let } \mathit{choices\text{-}trans} =\\
\qquad\quad \bigcup \{\mathit{build\text{-}transitions}(\mathit{mapify}(\mathit{rch\text{-}list}[i], -1), \mathit{poss\text{-}alt\text{-}maps}, k + (i-1)s)\\
\qquad\qquad \mid i \in \textbf{inds}\, \mathit{rch\text{-}list}\}\\
\quad \textbf{in}\\
\quad \textbf{let } \mathit{choices\text{-}input\text{-}maps} = \{\mathit{ivals}(\mathit{tr}) \mid \mathit{tr} \in \mathit{choices\text{-}trans}\} \;\; \textbf{in}\\
\quad \textbf{let } \mathit{else\text{-}maps} = \{m \in \mathit{poss\text{-}chooser\text{-}maps} \mid \neg\, \exists i \in \mathit{choices\text{-}input\text{-}maps} \cdot i \subseteq m\} \;\; \textbf{in}\\
\quad \textbf{let } \mathit{else\text{-}trans} =\\
\qquad\quad \bigcup \{\mathit{build\text{-}transitions}(m, \mathit{poss\text{-}alt\text{-}maps}, k + n \times s) \mid m \in \mathit{else\text{-}maps}\} \;\; \textbf{in}\\
\quad \mathit{mk\text{-}IOA}(\{0\}, \mathit{input\text{-}set}, \mathit{output\text{-}set}, \mathit{choices\text{-}trans} \cup \mathit{else\text{-}trans}, \{0\})
\end{array}
$$

$$
\begin{array}{l}
\mathit{build\text{-}transitions} : (\mathit{Name} \xrightarrow{m} \mathit{ChoiceVal}) \times (\mathbb{Z} \xrightarrow{m} \mathit{Val})\text{-}\textbf{set} \times \mathbb{N} \to \mathit{Trans}\text{-}\textbf{set}\\[1ex]
\mathit{build\text{-}transitions}(\mathit{chooser\text{-}in\text{-}map}, \mathit{in\text{-}maps}, \mathit{args\text{-}shift}) \quad \triangleq\\
\quad \{\mathit{mk\text{-}Trans}(0, \mathit{chooser\text{-}in\text{-}map} \cup (m \circ \mathit{shift}(\mathit{args\text{-}shift})), m \circ \mathit{reflect}, 0) \mid m \in \mathit{in\text{-}maps}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{mapify} : \mathit{RowChoiceVal} \times \mathbb{Z} \to (\mathbb{Z} \xrightarrow{m} \mathit{Val})\\[1ex]
\mathit{mapify}(\mathit{rcv}, \mathit{porm}) \quad \triangleq \quad \{\mathit{porm} \times i \mapsto \mathit{rcv}[i] \mid i \in \textbf{inds}\, \mathit{rcv} \wedge \mathit{rcv}[i] \neq X\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{shift} : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})\\[1ex]
\mathit{shift}(\mathit{sh})(n) \quad \triangleq \quad n - \mathit{sh}
\end{array}
$$

$$
\begin{array}{l}
\mathit{reflect} : \mathbb{Z} \to \mathbb{Z}\\[1ex]
\mathit{reflect}(n) \quad \triangleq \quad -n
\end{array}
$$

$$
\begin{array}{l}
\mathit{iter\text{-}causal\text{-}prod} : \mathit{IOA}^{+} \to \mathit{IOA}\\[1ex]
\mathit{iter\text{-}causal\text{-}prod}(\mathit{ioal}) \quad \triangleq \quad
\begin{array}[t]{l}
\textbf{cases}\ \mathit{ioal}\ \textbf{of}\\
\quad [\mathit{ioa}] \qquad\quad \to \mathit{ioa}\\
\quad [\mathit{ioa}] \frown \mathit{rest} \to \mathit{causal\text{-}product}(\mathit{ioa}, \mathit{iter\text{-}causal\text{-}prod}(\mathit{rest}))\\
\textbf{end}
\end{array}
\end{array}
$$

$get\text{-}choosers : CaseItem^* \to RowChoiceVal^*$

$$
\begin{array}{l}
get\text{-}choosers(ci\text{-}list) \quad \triangleq \\
\quad \textbf{cases}\ ci\text{-}list\ \textbf{of} \\
\quad [\,] \qquad\qquad \to [\,] \\
\quad [ci] \frown rest \to [chooser(ci)] \frown get\text{-}choosers(rest) \\
\quad \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
get\text{-}choices\text{-}rename\text{: } (init\text{-}shift\text{: }\mathbb{N}, out\text{-}len\text{: }\mathbb{N}, ci\text{-}list\text{: }CaseItem^+, \sigma\text{: }Env) \to ioa\text{-}list\text{: }IOA^+ \\
\textbf{post}\ (\textbf{len}\ ioa\text{-}list = \textbf{len}\ ci\text{-}list) \wedge \\
\quad \forall i \in \textbf{inds}\ ci\text{-}list \cdot \\
\quad\quad ioa\text{-}list[i] = rename(MExpression[\![result(ci\text{-}list[i])]\!]\sigma, outseq(init\text{-}shift, i-1, out\text{-}len))
\end{array}
$$

$$
\begin{array}{l}
outseq\text{: } (first\text{: }\mathbb{N}1, blocks\text{: }\mathbb{N}, length\text{: }\mathbb{N}1) \to os\text{: }\mathbb{Z}^* \\
\textbf{post len}\ os = length \wedge \\
\quad \forall i \in \textbf{inds}\ os \cdot \\
\quad\quad os[i] = -(first + blocks \times length + i)
\end{array}
$$

H.1.3.6 Function application

$$
\begin{array}{lll}
Funapp :: & func & : Name \\
 & args & : Expression^*
\end{array}
$$

$MFunapp : Funapp \to Env \to SemObj$

$$MFunapp[\![mk\text{-}Funapp(f, args)]\!]\sigma \quad \triangleq \quad apply(\sigma(f), eval\text{-}list(args, \sigma))$$

$eval\text{-}list : Expression^* \times Env \to Env$

$$
\begin{array}{ll}
eval\text{-}list(exps, \sigma) \quad \triangleq & \textbf{cases}\ exps\ \textbf{of} \\
 & [\,] \qquad\quad \to [\,] \\
 & [e] \frown rest \to [MExpression[\![e]\!]\sigma] \frown eval\text{-}list(rest, \sigma) \\
 & \textbf{end}
\end{array}
$$

$apply : SemObj \times SemObj^* \to SemObj$

$$
\begin{array}{l}
apply(fo, alist) \quad \triangleq \\
\quad restrict(bind([([1, \ldots, \textbf{card}\ outputs(fo)], fo)] \frown number\text{-}autlist(alist)), outputs(fo))
\end{array}
$$

$number\text{-}autlist : SemObj^* \to (Name^+ \times SemObj)^*$

$$
\begin{array}{ll}
number\text{-}autlist(l) \quad \triangleq & \textbf{cases}\ l\ \textbf{of} \\
 & [\,] \qquad\quad \to [\,] \\
 & l_1 \frown [a] \to number\text{-}autlist(l_1) \frown [([-\textbf{len}\ l], a)] \\
 & \textbf{end}
\end{array}
$$

$restrict : IOA \times Name\text{-}\mathbf{set} \to SemObj$

$restrict(a, xs) \quad \triangleq$
  $\quad \mathbf{let}\ new\text{-}ts = \{\mu(t, ovals \mapsto \{x \mapsto ovals(t)(x)\}) \mid t \in ts(a) \land x \in xs\}\ \ \mathbf{in}$
  $\quad \mu(a, transitions \mapsto new\text{-}ts)$

$\mathbf{pre}\ xs \subseteq outputs(a)$

H.1.3.7 Delays

H.1.3.7.1 Inertial delay

$Delays = IDelay \mid Delay$

$RowOfVal = Val^+$

$$\begin{aligned} IDelay\ ::\ \ arg &: Expression \\ const &: RowOfVal \\ int &: \mathbb{N} \end{aligned}$$

$MIDelay : IDelay \times Env \to SemObj$

$MIDelay[\![mk\text{-}IDelay(e, c, i)]\!]\sigma \quad \triangleq$
  $\quad \mathbf{let}\ exp\text{-}obj = MExpression[\![e]\!]\sigma\ \ \mathbf{in}$
  $\quad \mathbf{let}\ w = \mathbf{card}\ outputs(exp\text{-}obj)\ \ \mathbf{in}$
  $\quad apply(mk\text{-}idel(c, i, w), exp\text{-}obj)$

$mk\text{-}idel : RowOfVal \times \mathbb{N} \times Nat_1 \to SemObj$

$mk\text{-}idel(c, i, w) \quad \triangleq$
  $\quad \mathbf{let}\ states = \{s \frown t \in RowOfVal^* \mid \mathbf{len}\ s \frown t = i,$
  $\qquad\qquad \mathbf{elems}\ s = \{a_1\}, \mathbf{elems}\ t = \{a_2\}, a_1 \in RowOfVal, a_2 \in RowOfVal\}\ \ \mathbf{in}$
  $\quad \mathbf{let}\ inits = \{s \in states \mid \mathbf{elems}\ s = \{c\}\}\ \ \mathbf{in}$
  $\quad \mathbf{let}\ ts = \{mk\text{-}Trans(p, mapify(v, -1), mapify(\mathbf{hd}\ p, 1), q) \mid$
  $\qquad\qquad p \in states, v \in RowOfVal, q = choose\text{-}state(p, v)\}\ \ \mathbf{in}$
  $\quad mk\text{-}IOA(states, \{-w, \ldots, -1\}, \{1, \ldots, w\}, ts, inits)$

$\mathbf{pre}\ \mathbf{len}\ c = w$

$choose\text{-}state : RowOfVal^+ \times RowOfVal \to RowOfVal^+$

$choose\text{-}state(s, a) \quad \triangleq \quad \mathbf{if}\ s(\mathbf{len}\ s) = a$
  $\qquad \mathbf{then}\ \mathbf{tl}\ s \frown [a]$
  $\qquad \mathbf{else}\ \mathbf{let}\ n = (\mathbf{len}\ s)\text{-}1\ \ \mathbf{in}$
  $\qquad\quad [\mathbf{hd}\ s]^n \frown [a]$

H.1.3.7.2 Ambiguity Delay

$RowOfVal = Val^+$

$$\begin{aligned} Delay\ ::\ \ arg &: Expression \\ const_1 &: RowOfVal \\ int_1 &: \mathbb{N} \\ const_2 &: RowOfVal \\ int_2 &: \mathbb{N} \end{aligned}$$

$MDelay : Delay \times Env \to SemObj$

$MDelay[\![mk\text{-}Delay(e, c_1, i_1, c_2, i_2)]\!]\sigma \quad \triangleq$
$\quad$ **let** $exp\text{-}obj = MExpression[\![e]\!]\sigma$ **in**
$\quad$ **let** $w = \mathbf{card}\, outputs(exp\text{-}obj)$ **in**
$\quad$ $apply(mk\text{-}del(c_1, i_1, c_2, i_2, w), exp\text{-}obj)$

$mk\text{-}del : RowOfVal \times \mathbb{N} \times RowOfVal \times \mathbb{N} \times Nat_1 \to SemObj$

$mk\text{-}del(c_1, i_1, c_2, i_2, w) \quad \triangleq$
$\quad$ **let** $states = \{s \in RowOfVal^* \mid \mathbf{len}\, s = i_1 + i_2\text{-}1\}$ **in**
$\quad$ **let** $inits = \{s \in states \mid \mathbf{elems}\, s = \{c_1\}\}$ **in**
$\quad$ **let** $ts = \{mk\text{-}Trans(p, mapify(v, -1), mapify(\mathbf{hd}\, p, 1), \mathbf{tl}\, p \frown [v]) \mid$
$\quad\quad\quad p \in states, q = next\text{-}output(p, i_1, i_2, c_2), v \in RowOfVal\}$ **in**
$\quad$ $mk\text{-}IOA(states, \{-w, \ldots, -1\}, \{1, \ldots, w\}, ts, inits)$

pre $\mathbf{len}\, c_1 = w \wedge \mathbf{len}\, c_2 = w$

$next\text{-}output : RowOfVal^* \times \mathbb{N} \times \mathbb{N} \times RowOfVal \to RowOfVal$

$next\text{-}output(s, m, n, c) \quad \triangleq \quad$ **if** $subsequence(s, n)$ **then** $s(m)$ **else** c

$subsequence : RowOfVal^* \times \mathbb{N} \to \mathbb{B}$

$subsequence(s, n) \quad \triangleq \quad \exists s_1, s_2 \in RowOfVal^*, v \in RowOfVal \cdot s = s_1 \frown [v]^n \frown s_2$

H.1.3.7.3 Pure delay $Delay_{pure}$ $\quad RowOfVal = Val^+$

$$\begin{array}{lrcl} Delay_{pure} :: & arg & : & Expression \\ & const & : & RowOfVal \\ & int & : & \mathbb{N} \end{array}$$

$MDelay_{pure} : Delay_{pure} \times Env \to SemObj$

$MDelay_{pure}[\![mk\text{-}Delay_{pure}(e, c, i)]\!]\sigma \quad \triangleq$
$\quad$ **let** $exp\text{-}obj = MExpression[\![e]\!]\sigma$ **in**
$\quad$ **let** $w = \mathbf{card}\, outputs(exp\text{-}obj)$ **in**
$\quad$ $apply(mk\text{-}del_{pure}(c, i, w), exp\text{-}obj)$

$mk\text{-}del_{pure} : RowOfVal \times \mathbb{N} \times Nat_1 \to SemObj$

$mk\text{-}del_{pure}(c, i, w) \quad \triangleq$
$\quad$ **let** $states = \{s \in RowOfVal^* \mid \mathbf{len}\, s = i\}$ **in**
$\quad$ **let** $inits = \{s \in states \mid \mathbf{elems}\, s = \{c\}\}$ **in**
$\quad$ **let** $ts = \{mk\text{-}Trans(p, mapify(v, -1), mapify(\mathbf{hd}\, p, 1), (\mathbf{tl}\, p) \frown [v]) \mid$
$\quad\quad\quad p \in states, v \in RowOfVal\}$ **in**
$\quad$ $mk\text{-}IOA(states, \{-w, \ldots, -1\}, \{1, \ldots, w\}, ts, inits)$

pre $\mathbf{len}\, c = w$

H.1.3.7.4 Transport Delay *trans* $\quad RowOfVal = Val^+$

$$\begin{array}{rrcl} Delay_{trans} & :: \quad arg & : & Expression \\ & const & : & RowOfVal \\ & int & : & \mathbb{N} \end{array}$$

$MDelay_{trans} : Delay_{trans} \times Env \to SemObj$

$$\begin{array}{l} MDelay_{trans}[\![mk\text{-}Delay_{trans}(e, c, i)]\!]\sigma \quad \triangleq \\ \quad \textbf{let } exp\text{-}obj = MExpression[\![e]\!]\sigma \textbf{ in} \\ \quad \textbf{let } w = \textbf{card } outputs(exp\text{-}obj) \textbf{ in} \\ \quad apply(mk\text{-}del_{trans}(c, i, w), exp\text{-}obj) \end{array}$$

$mk\text{-}del_{trans} : RowOfVal \times \mathbb{N} \times \mathbb{N}_1 \to SemObj$

$$\begin{array}{l} mk\text{-}del_{trans}(c, i, w) \quad \triangleq \\ \quad \textbf{let } states = \{s \mid s = [f]^k \frown [g]^l \frown [h]^m \wedge \textbf{len } s = i \wedge f, g, h \in RowOfVal \wedge k, l, m \in \mathbb{N}\} \textbf{ in} \\ \quad \textbf{let } inits = \{s \in states \mid \textbf{elems } s = \{c\}\} \textbf{ in} \\ \quad \textbf{let } ts = \{mk\text{-}Trans(p, mapify(v, -1), mapify(\textbf{hd } p, 1), q) \mid \\ \qquad\qquad p \in states, v \in RowOfVal, q = next\text{-}state(p, v, c, i)\} \textbf{ in} \\ \quad mk\text{-}IOA(states, \{-w, \ldots, -1\}, \{1, \ldots, w\}, ts, inits) \end{array}$$

$next\text{-}state : RowOfVal^* \times RowOfVal \times \mathbb{N} \to RowOfVal^*$

$$\begin{array}{ll} next\text{-}state(p, v, c, i) \quad \triangleq & \textbf{if } p(\textbf{len } p) = v \\ & \textbf{then tl } p \frown [v] \\ & \textbf{else let } a = \textbf{hd } p \textbf{ in} \\ & \quad \textbf{let } j = \iota j \in \mathbb{N} \cdot [a]^j \frown s = p \wedge \textbf{hd } s \neq a \textbf{ in} \\ & \quad [a]^{j\text{-}1} \frown [c]^{i\text{-}j} \frown [v] \end{array}$$

H.1.3.8 Sample

$$\begin{array}{rrcl} Sample & :: \quad expr & : & Expression \\ & const & : & Val^+ \\ & skew & : & \mathbb{N} \\ & int & : & \mathbb{N}_1 \end{array}$$

$MSample : Expression \times Val^+ \times \mathbb{N} \times \mathbb{N}_1 \to SemObj$

$$\begin{array}{l} MSample[\![expr, const, skew, int]\!]\sigma \quad \triangleq \\ \quad \textbf{let } exp\text{-}obj = MExpression[\![expr]\!]\sigma \textbf{ in} \\ \quad \textbf{let } width = \textbf{card } outputs(exp\text{-}obj) \textbf{ in} \\ \quad apply(Sampler(const, skew, int, width), exp\text{-}obj) \end{array}$$

$Sampler : Val^{+} \times \mathbb{N} \times \mathbb{N}_1 \times \mathbb{N}_1 \to SemObj$

$$
\begin{array}{l}
Sampler(c,s,n,w) \quad \triangleq \\
\quad \textbf{let } inits = \{-s\} \ \textbf{ in} \\
\quad \textbf{let } skews = \{-s+i \mid i \in \{1,\ldots,s\}\} \ \textbf{ in} \\
\quad \textbf{let } cyclers = \{(v,i) \mid v \in Val^{w} \wedge i \in \{2,\ldots,n-1\}\} \ \textbf{ in} \\
\quad \textbf{let } states = inits \cup skews \cup cyclers \ \textbf{ in} \\
\quad \textbf{let } ts = \{mk\text{-}Trans(-s+j,\{\,\},mapify(c,1),-s+j+1) \mid j \in \{0,\ldots,s-1\}\} \cup \\
\qquad\qquad \{mk\text{-}Trans(0,mapify(v,-1),mapify(v,1),0) \mid v \in Val^{w} \wedge n = 1\} \cup \\
\qquad\qquad \{mk\text{-}Trans(0,mapify(v,-1),mapify(v,1),(v,1)), \\
\qquad\qquad\ \ mk\text{-}Trans((v,n-1),\{\,\},mapify(v,1),0) \mid v \in Val^{w} \wedge n > 1\} \cup \\
\qquad\qquad \{mk\text{-}Trans((v,i),\{\,\},mapify(v,1),(v,i+1)) \\
\qquad\qquad\qquad \mid v \in Val^{w} \wedge i \in \{1,\ldots,n-2\}\} \ \textbf{ in} \\
\quad mk\text{-}IOA(states,\{-w,\ldots,-1\},\{1,\ldots,w\},ts,inits)
\end{array}
$$

$\textbf{pre len}\, c = w$

H.1.3.9 BIOPS: Built-in Operators

$Not \ :: \ negated \ : \ Expression$

$MNot : Not \to Env \to SemObj$

$MNot[\![mk\text{-}Not(n)]\!]\sigma \quad \triangleq \quad apply(not\text{-}obj, MExpression[\![n]\!]\sigma)$

$Binop = \mathrm{OR} \mid \mathrm{AND}$

$$
\begin{array}{lrcl}
Infix \ :: & arg1 & : & Expression \\
& op & : & Binop \\
& arg2 & : & Expression
\end{array}
$$

$MInfix : Infix \to Env \to SemObj$

$$
\begin{array}{l}
MInfix[\![mk\text{-}InFix(e_1,o,e_2)]\!]\sigma \quad \triangleq \\
\quad apply(MBinop[\![o]\!],[MExpression[\![e_1]\!]\sigma, MExpression[\![e_2]\!]\sigma])
\end{array}
$$

H.1.4 Function declarations

$$
\begin{array}{lrcl}
FnDec \ :: & name & : & Id \\
& params & : & Id^{*} \\
& body & : & Expression
\end{array}
$$

$MFnDec : FnDec \to Env \to Env$

$$
\begin{array}{l}
MFnDec[\![mk\text{-}FnDec(n,ps,b)]\!]\sigma \quad \triangleq \\
\quad \textbf{let } obj\text{-}body = number\text{-}inputs(MExpression[\![b]\!]\sigma, ps) \ \textbf{ in} \\
\quad \sigma \dagger \{f \mapsto obj\text{-}body\}
\end{array}
$$

$number\text{-}inputs : IOA \times Id^* \to IOA$

$$
\begin{array}{l}
number\text{-}inputs(a, ids) \quad \triangleq \\
\quad \textbf{let } mk\text{-}IOA(ss, i, o, ts, start) = a \ \textbf{ in} \\
\quad \textbf{let } subst = make\text{-}subst(i, ids) \ \textbf{ in} \\
\quad \textbf{let } new\text{-}ts = \{\mu(t, ivals \mapsto ivals(t) \circ subst) \mid t \in ts\} \ \textbf{ in} \\
\quad mk\text{-}IOA(ss, \textbf{dom}\, subst, o, new\text{-}ts, start)
\end{array}
$$

$\textbf{pre } inputs(a) \subseteq Id$

$make\text{-}subst : Id\text{-}\textbf{set} \times Id^* \to (Name \xleftrightarrow{m} Name)$

$$
\begin{array}{l}
make\text{-}subst(iset, ilist) \quad \triangleq \\
\quad \textbf{let } inv = \{x \mapsto x \mid x \in iset\} \dagger \{ilist(k) \mapsto -k \mid ilist(k) \in iset \land k \in \textbf{inds}\, ilist\} \ \textbf{ in} \\
\quad inv^{-1}
\end{array}
$$

H.1.5 Closures

$Closure = FnDec^+$

$MClosure : Closure \to Env$

$MClosure[\![ds]\!] \quad \triangleq \quad MFnDecList[\![ds]\!]$

$MFnDecList : FnDec^* \to Env$

$$
MFnDecList[\![ds]\!] \quad \triangleq \quad
\begin{array}{l}
\textbf{cases } ds \ \textbf{ of} \\
[\,] \qquad\quad \to \{\,\} \\
l \frown [dec] \to MFnDec[\![dec]\!](MFnDecList[\![l]\!]) \\
\textbf{end}
\end{array}
$$

References

[Bar91] H Barringer.
Private communication, 1991.

[BC86] M.C. Browne and E.M. Clarke.
SML - a high level language for the design and verification of finite state machines.
In *From H.D.L. descriptions to guaranteed correct circuit designs*. IFIP, September 1986.

[Bec87] A. Beck.
Using temporal logic to verify properties of circuit designs described in ELLA.
Master's thesis, St. Cross College, Oxford University, September 1987.

[BEGS89] H. Barringer, D. Edwards, J. Gurd, and V. Stavridou.
The HAVE project final report.
University of Manchester Computer Science Department, 1989.

[BFG+89a] H Barringer, M Fisher, D Gabbay, G Gough, and R Owens.
METATEM: A Framework for Programming in Temporal Logic.
Technical Report UMCS-89-10-4, Department of Computer Science, University of Manchester, 1989.

[BFG89b] H. Barringer, M. Fisher, and G.D. Gough.
Fair SMG and Linear Time Model Checking.
In *Proceedings of Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.
(Published in *Lecture Notes in Computer Science* volume 407).

[BGHT90] R Boulton, M Gordon, J Herbert, and J Van Tassel.
The HOL Verification of ELLA Designs.
Technical report, University of Cambridge Computer Laboratory, 1990.

[BGM91] H. Barringer, G. Gough, and B. Monahan.
Operational Semantics of Hardware Design Languages.
Technical Report UMCS-91-2-2, University of Manchester Computer Science Department, February 1991.

[BHM90] H. Barringer, M. Hill, and B. Monahan.
Towards an Operational Semantics of Ella.
Technical Report 1.1a, Formal Verification Support for Ella, IED project 4/1/1357, October 1990.

[CE82] E.M. Clarke and E.A. Emerson.
Synthesis of synchronization skeletons from branching time temporal logic.
In *Proceedings of the Workshop on Logics of Programs*, pages 52–71, Berlin, 1982. Volume 131, LNCS, Springer-Verlag.

[CES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla.
Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications: A Practical Approach.
In *Proceedings of the Tenth ACM Symposium on the Principles of Programming Languages*, 1983.

[CGM86] A Camilleri, M Gordon, and T Melham.
Hardware Verification using Higher-Order Logic.

Technical report, University of Cambridge Computer Laboratory, June 1986.

[Coh87] A Cohn.
A Proof of Correctness of the VIPER Microprocessor: The First Level.
Technical report, University of Cambridge Computer Laboratory, January 1987.

[Coh89] A Cohn.
The Notion of Proof in Hardware Verification.
Journal of Automated Reasoning, 5, 1989.

[Com90a] Computer General Electronic Design, The New Church, Henry Street, Bath, Avon, BA1 1JR, United Kingdom.
The ELLA Language Reference Manual, 4.0th edition, 1990.

[Com90b] Computer General Electronic Design, The New Church, Henry Street, Bath, Avon, BA1 1JR, United Kingdom.
The ELLA Tutorial, 4.0th edition, 1990.

[Det86] R. Dettmer.
Ella - a language for VLSI.
Electronics and Power, pages 517–522, July 1986.

[Gab91] Dov Gabbay.
Temporal logic — mathematical foundations and computational aspects.
To appear, 1991.

[GB88] G. D. Gough and H. Barringer.
A Semantics Driven Temporal Verification System.
In *Proceedings of ESOP '88*. LNCS 300, Springer-Verlag, March 1988.

[Gog88] J A Goguen.
OBJ as a Theorem Prover with Application to Hardware Verification.
Technical report, SRI International, August 1988.

[Goo90a] K.G.W. Goossens.
An Operational Semantics for a Subset of the CHDDL ELLA.
Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, April 1990.

[Goo90b] K.G.W. Goossens.
Semantics for picoELLA.
Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, June 1990.

[GW88] J A Goguen and T Winkler.
Introducing OBJ3.
Technical report, SRI International, August 1988.

[Hen88] M. Hennessy.
Algebraic Theory of Processes.
MIT Press, 1988.

[Hen90] M. Hennessy.
The Semantics of Programming Languages.
John Wiley & Sons, 1990.

[HK90] Z Har'El and R P Kurshan.
Software for Analytical Development of Communications Protocols.
AT&T Technical Journal, January/February 1990.

[HM91] M.G. Hill and J.D. Morision.
Preliminary Core ELLA Definition.
Internal Memorandum 1.1c, Formal Verification Support for Ella, IED project 4/1/1357, February 1991.

[HMT90] R. Harper, R. Milner, and M. Tofte.
The Definition of Standard ML.
MIT Press, 1990.

[HPC+90] M.G. Hill, N.E. Peeling, I.F. Currie, J.D. Morison, E.V. Whiting, and C.O. Newton.
Real number arithmetic for mixed behavioural and structural descriptions.
Proceedings of the IEE, 137(G):446–450, 1990.

[HU79a] J.E. Hopcroft and J.D. Ullman.
Introduction to Automata Theory, Languages and Computation.
Addison Wesley, 1979.

[HU79b] J.E. Hopcroft and J.D. Ullman.
Introduction to Automata Theory, Languages and Computation.
Addison Wesley, 1979.

[HW89] M.G. Hill and E.V. Whiting.
Ella timing.
Technical Report D3.3/RSRE/3, ESPRIT project 2260 (SPRITE), 1989.

[HWM90a] M.G. Hill, E.V. Whiting, and J.D. Morison.
Formal semantic definition of ella timing.
Internal Memorandum 4436, Royal Signals and Radar Establishment, Great Malvern, 1990.

[HWM90b] M.G. Hill, E.V. Whiting, and J.D. Morison.
Formal Semantic Definition of ELLA Timing.
Internal Memorandum Memo 4436, Royal Signals and Radar Establishment, November 1990.

[HWM90c] M.G. Hill, E.V. Whiting, and J.D. Morison.
Sprite-ella language enhancements.
Internal Memorandum 4441, Royal Signals and Radar Establishment, Great Malvern, 1990.

[HWM91] M.G. Hill, E.V. Whiting, and J.D. Morison.
Bidirectionality, connectivity and instantiations in ella.
Internal Memorandum 4421, Royal Signals and Radar Establishment, Great Malvern, 1991.

[Jon90] Cliff B. Jones.
Systematic Software Development using VDM.
Prentice Hall, second edition, 1990.

[JS90] Cliff B. Jones and R. C. Shaw.
Case Studies in Systematic Software Development.

Prentice Hall, 1990.

[Ker84] J Kershaw.
VIPER: A Microprocessor for Safety-Critical Application.
Internal memorandum no. 3754, Royal Signals and Radar Establishment, Great Malvern, December 1984.

[LT88] N.A. Lynch and M.R. Tuttle.
An Introduction to Input/Output Automata.
Technical Report MIT/LCS/TM-373, MIT, November 1988.

[Mil90] R. Milner.
Concurrency and Communication.
Prentice Hall, 1990.

[Mon91] B.Q. Monahan.
A note on the semantics of ambiguity.
Project working paper, University of Manchester Computer Science Department, January 1991.

[Moo56] E.F. Moore.
Gedanken Experiments on Sequential Machines.
In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 129 – 153. Princeton University Press, 1956.

[Mos85] B. Moszkowski.
A Temporal Logic for Multilevel Reasoning About Hardware.
IEEE Computer, February 1985.

[MPT85a] J.D. Morison, N.E. Peeling, and T.L. Thorp.
The design rationale of Ella, a hardware design and description language.
In *CHDL*, Tokyo, 1985.

[MPT85b] J.D. Morison, N.E. Peeling, and T.L. Thorp.
The design rationale of Ella, a hardware design and description language.
In *CHDL*, Tokyo, 1985.

[Pit88] E B Pitty.
A Critique of the GATEMAP Logic Synthesis System.
In *International Workshop on Logic and Architecture Synthesis For Silicon Compilers, Grenoble*, May 1988.

[Plo81] G. Plotkin.
A structural approach to operational semantics.
Technical Report FN-19, DAIMI, Sept 1981.

[RRSV87] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron.
Xesar a tool for protocol verification: User's guide.
Technical report, Laboratoire de Génie Informatique de Grenoble, IMAG-Campus, B.P.53H, 38041 Grenoble Cedex, France, September 1987.

[RS59] M.O. Rabin and D. Scott.
Finite Automata and their Decision Problems.
IBM Journal of Research and Development, 3(2):114–125, April 1959.

[SBE90] V. Stavridou, H. Barringer, and D. Edwards.
Formal Specification and Verification of Hardware: A Comparative Case Study.
In Michael Yoeli, editor, *Formal Verification of Hardware Design*, chapter 4, pages 89–96. IEEE Computer Society, 1990.

[Sch67] J.R. Schoenfield.
Mathematical Logic.
Addison Wesley, 1967.

[Shi87] M. Shields.
An Introduction to Automata Theory.
Blackwell, Oxford, 1987.

[Tai88a] S. Tait.
Formal specification of built in operations.
Internal Memorandum P209.40.5, Praxis, 1988.

[Tai88b] S. Tait.
System specification for a new simulator.
Internal Memorandum P209.40.4, Praxis, 1988.

[TH89] J Van Tassel and David Hemmendinger.
Toward Formal Verification of VHDL Specifications.
In *Applied Formal Methods For Correct VLSI Design*. IMEC-IFIP Workshop, 1989.

[Tho90] W. Thomas.
Handbook of Theoretical Computer Science, volume B, chapter 4, pages 133 – 192.
Elsevier, 1990.
(ed: Jan Van Leeuwen).

[VW86] M.Y. Vardi and P. Wolper.
Automata Theoretic techniques for Modal Logics of Programs.
Journal of Computer and System Sciences, 32(2):183–221, April 1986.

[VW88] M.Y. Vardi and P. Wolper.
Reasoning about Infinite Computation Paths.
Technical Report RJ 6209 (61199), IBM Research Division, April 1988.

[Wil90] A Williams.
Verification Requirements Analysis.
Technical Report 2.1a, Formal Verification Support for ELLA, IED Project 4/1/1357, Department of Computer Science, University of Manchester, September 1990.

[Wol89] P. Wolper.
From Modal Logic to Deductive Databases, chapter 4, pages 165 – 234.
John Wiley & Sons, 1989.
(ed: A.Thayse).