Diagrammatic Representations in Domain-Specific Languages

Konstantinos Tourlas

Doctor of Philosophy
University of Edinburgh
2001

Abstract

One emerging approach to reducing the labour and costs of software development favours the specialisation of techniques to particular application domains. The rationale is that programs within a given domain often share enough common features and assumptions to enable the incorporation of substantial support mechanisms into domain-specific programming languages and associated tools.

Instead of being machine-oriented, algorithmic implementations, programs in many domain-specific languages (DSLs) are rather user-level, problem-oriented *specifications* of solutions. Taken further, this view suggests that the most appropriate representation of programs in many domains is *diagrammatic*, in a way which derives from existing design notations in the domain.

This thesis conducts an investigation, using mathematical techniques and supported by case studies, of issues arising from the use of diagrammatic representations in DSLs. Its structure is conceptually divided into two parts: the first is concerned with semantic and reasoning issues; the second introduces an approach to describing the syntax and layout of diagrams, in a way which addresses some pragmatic aspects of their use.

The empirical context of our work is that of IEC 1131-3, an industry standard programming language for embedded control systems. The diagrammatic syntax of IEC 1131-3 consists of circuit (i.e. box-and-wire) diagrams, emphasising a data-flow view, and variants of Petri net diagrams, suited to a control-flow view.

The first contribution of the thesis is the formalisation of the diagrammatic syntax and the semantics of IEC 1131-3 languages, as a prerequisite to the application of algebraic techniques. More generally, we outline an approach to the design of diagrammatic DSLs, emphasising compositionality in the semantics of the language so as to allow the development of simple proof systems for inferring properties which are deemed essential in the domain. The control-flow subset of IEC 1131-3 is carefully evaluated, and is subsequently re-designed, to yield a straightforward proof system for a restricted, yet commonly occurring, class of safety properties.

A substantial part of the thesis deals with DSLs in which programs may be represented both textually and diagrammatically, as indeed is the case with IEC 1131-3. We develop a formalisation of the data-flow diagrams in IEC 1131-3 and rigorously demonstrate the match between these diagrams, their internal textual equivalents and their computational interpretation. Using the devices of universal algebra, we formulate a general, rigorous consistency criterion regarding such “dual” representations.

Finally, we motivate the need to include pragmatic aspects of diagrams, such as layout, in formalisations. We demonstrate how certain classes of categories can capture both the structure and layout of graph-based notations. Use of this technique is concretely illustrated in terms of control-flow diagrams in IEC 1131-3, yielding a basis on which diagram-driven and layout-sensitive rules of inference can be formulated.

Acknowledgements

To Stuart Anderson, my supervisor, I owe a debt of gratitude, not least for his patience and skillful handling of my perpetually oscillating mood, which often swings from optimism to pessimism in a matter of minutes. Stuart has had a profound influence on my outlook to computing, instilling in me a genuinely holistic view: that understanding computing at the social and cognitive levels is as important, and complementary to, any technological or mathematical understanding. After all, computer systems are designed for and used by people.

It was Stuart, too, who introduced me to Corin Gurr, with whom I've had great pleasure working and from whom I have learned a great deal. Substantial parts of this thesis have, directly or indirectly, benefited from our collaboration, which I hope continues in the future.

During my third year, I was privileged to receive interest and encouragement from John Power. John's highly pragmatic advice, always accompanied by his unique sense of humour, has had enormous influence on my view of the research process.

My warmest thanks, too, to my friends Carsten, Conor, Pauline and Steve for an endless supply of much needed displacement activities, useful discussions and a great time.

Declaration

I declare that this thesis was composed by myself and that the work contained herein is my own, with the exception of the material in Chapter 6 which is the product of joint work with Dr Corin A. Gurr[1]. Early versions of parts of this thesis have already been published elsewhere:

- Chapter 3 as [49] (sections 3.4 and 3.5) and [5, 6] (rest of chapter)
- Chapter 4 as [4]; and
- Chapter 6 as [48].

Konstantinos Tourlas

[1] Corin Gurr is a member of the Human Communication Research Centre (HCRC) at the University of Edinburgh.

Table of Contents

Chapter 1

Introduction

There is a tension between engineering practice and the objectives of programmability in computer systems. Typically, engineering practice is *domain-specific* exploiting the constraints of the class of systems under study to simplify the design task for the engineer. By contrast, programmable systems rest upon the principles of universality, generality and ultimate flexibility. The recent popularisation of *domain-specific programming languages* propounds one approach to resolving this tension.

Our view emphasises representations for programmable systems that incorporate domain knowledge and notation. In many situations domain notations are extensively diagrammatic, thus suggesting a naturally diagrammatic syntax for programming languages specific to the domain. This incorporation of domain notation in program representations provides strong support for the integration of programmable components into the engineering practices of domains. However, there is as yet no systematic study of the qualities and features representations must possess in order to be effective in supporting this integration.

To this end, this thesis contributes the novel application of established and emerging mathematical techniques to the analysis of diagrammatic domain-specific languages. Our analysis is not only based on languages enjoying wide industrial use, but also motivated by pragmatic issues arising from the use of diagrams in such languages. One issue relates to the potential divergence between a user's interpretation of diagrams as artifacts in the domain and their actual computational interpretation. The other issues addressed concern the degree to which diagrammatic syntax supports domain experts in reasoning about programs and the use of secondary features (such as layout) in encoding domain knowledge.

1.1 The practical problem

Seldom is any domain of knowledge or expertise mature without the development of specialised vocabulary and notations for the description and analysis of artifacts in the domain. One one hand, the role of notations is cognitive in nature as they provide support for basic design and reasoning tasks. On the other hand, notations also have an important social role as communication interfaces between different, and possibly diverse, technical specialities involved in a domain.

To date, Computing Science has focused primarily on universal notations for system representation which are primitive and thus generic: general-purpose programming languages and hardware descriptions in terms of basic components (e.g. logic gates, counters, etc.) are both examples of such notations.

On the other hand the development of domain-specific techniques in computing has been remarkably slow—except perhaps in hardware design where building blocks are becoming increasingly application-specific. Rectifying this situation is therefore being argued for as a vital area of future research, as, for instance, in a document [98] by Bennett and McDermid submitted recently to the Engineering Board of the British Computer Society:

> *Work on artifacts, or products, has to be domain-specific, i.e. to reflect the structure of artifacts in a particular industry. [. . .] In other engineering disciplines, the underlying science is adapted to provide domain specific notations and analyses. It is important to adopt a similar approach in Software Engineering to ensure that research produces usable methods which offer significant technical advantage.*

The design of domain-specific programming languages also features prominently in [45], a report to the ACM meeting on Strategic Directions in Computing Research (organised on the Association's 50th anniversary), as one of four areas that demand additional research immediately. According to Gunter, Mitchell and Notkin:

> *Domain-specific languages are intended to allow domain engineers to develop families of applications that are easily specified, highly evolvable, and largely automated.*

In the same report, the formalisation of informal notations found in domains and the development of theoretical frameworks applicable to domain-specific languages are proposed as specific sub-goals. These provide the starting point of our research.

1.2 The nature of our approach

One way of characterising a domain is in terms of the practices and design processes that experts in the domain employ; the objects of such practices and processes being the *artifacts* of interest in the domain. An essential feature of design processes in mature domains is the use of specialised notations to provide convenient representations of the artifacts being designed with the aim of facilitating analyses and the inference of commonly required properties. Control engineers, for instance, have developed variants of circuit diagrams and Petri nets as notations for the design and analysis of embedded controllers. Similarly in rail transport, engineers have evolved their own notation to represent the complex structure of signalling systems [107, 106].

Central to our approach is the belief that domain-specific representations of computer software should accord with, and ideally be derived from existing notations in the domain. As an instance of this, we consider the diagrammatic representation of software elements in the formalism of IEC 1131-3, the industry-standard programming language for embedded control systems. Software diagrams in IEC 1131-3 derive directly from circuit notations and, as such, they encourage thinking in terms of a "hardware" paradigm which is familiar to control engineers. In actuality, however, these diagrams are internally translated into sequential programs and executed by microprocessors.

This example illustrates how representations can support understanding of system structure and operation in terms of artifacts in the domain—in our case the hard-wired controller circuits which software elements replace. Our may therefore attempt to understand domains first at a *social* level, in terms of engineering practice, and at a *cognitive* level, in terms of the way representations support individuals contributing to engineering practice. Here, our work centres on the relation between representations and domain artifacts and how appropriate choices of representation ease reasoning tasks. Particular choices of representation may differ profoundly in their ability to support particular tasks and inferences. In Chapter 3, for example, we show how IEC 1131-3 diagrams could be improved to facilitate the formulation of a simple proof system for expressing and validating simple safety arguments. Our eventual aim, partly realised in this thesis, is to inform mathematical analyses of diagrammatic representations by the findings of established cognitive analyses.

In order to validate the use of diagrammatic representations as vehicles for the analysis and programming of domain-specific software, one requires a theory of how these representations relate to concrete implementations. Our second concern is the development of such a theory, based upon rigorous models of the abstract syntax of diagrams (static structure) and semantics (dynamic behaviour). It seems that diagrams aim to make concrete some *structural* aspects of systems while the *behaviour* of systems bears some (possibly complex) relation to the structure.

This distinction between structural and semantic (behavioural) models is common in the study of the semantics of computation where some structural equivalence is introduced to ease the presentation of the semantic models (e.g. CCS [103] and the pi-calculus [104]). Here we take the view that interposing a structural model between the concrete (and largely informal) representations and the semantic models provides an intermediate level that is suited to studying domain specific semantics and reasoning. We aim to study and develop techniques which, whilst rigorous and semantically sound, enable reasoning to be conducted primarily on representation and be justified by reference to structural models. An approach of how this can be done in a formally justifiable manner will be presented in Chapters 3 and 6.

Finally, our approach is concerned with criteria which representations impose on the suitability of candidate implementations. In Chapter 4, for example, we define an algebra corresponding to the abstract syntax of a class of IEC 1131-3 diagrams and formally demonstrate the match between these diagrams, their internal textual equivalents and their computational interpretation. The result generalises to a rigorous criterion for the use of diagrammatic notations in domain-specific languages.

However, the scope of the research herein presented is only partly contained within the well-established field of programming language semantics. More broadly, we focus on two areas that promise to forge a link between cognitive and semantic concerns in domain specific notations:

- the extent to which features identified by cognitive analysis of domain specific notations can be incorporated into expressive structural models.

- the study of mappings between structural and behavioural models in their capacity to justify the manipulation of diagrams by humans in reasoning about the behaviour of systems.

1.3 Related work and literature

1.3.1 Domain-specific languages

Recently there has been a resurgence within the programming languages community of interest in domain-specific languages (DSLs) [117, 76]. Among a majority of publications detailing and justifying the design of individual languages, there is a growing number of attempts to address issues of wider concern. Broadly, these issues relate to the implementation and methodological design of DSLs.

1.3.1.1 Implementation techniques

Investigations into the rapid implementation and prototyping of DSLs have yielded a number of approaches based on the specialisation of existing syntactic and semantic techniques. These range from the mostly syntactic (e.g. “jargons” [108]) to the mostly semantic (e.g. constraint programming [34, 35]). Among the most popular are techniques based on the embedding of DSLs into general-purpose languages and the domain-specific compiler or program specialisation.

The use of general-purpose languages as hosts for DSLs is advocated mainly by the groups led by Hudak [70] and Kamin [77]. In their approach, a newly devised DSL inherits the infrastructure of a high-level, strongly-typed functional language such as ML or Haskell. In addition to drastically cutting down implementation effort, this in principle enables reuse, or specialisation, of static analysis and type-checking tools. The main limitation, in our view, relates to the extent in which the DSL semantics is naturally expressible in terms of that of its host language. (The same argument is often used by the proponents of Haskell owing to the language’s ability to express a variety of user-defined semantic effects via monad transformations [87].) When an adequate semantic match exists, the results can be highly satisfactory as demonstrated by case studies in 3D graphics animation [29, 30], picture-drawing [78] and “geometric region servers” [19]. More recently, variants of the approach employ the host language to construct standalone, modular interpreters for DSLs. There appears to be some friction between the objectives of interpreter modularity and optimisation [71] and a variety of technologies (including staging, monads [122] and partial evaluation [71]) are currently being evaluated.

The other popular approach regards programs in a DSL as a family, i.e. a class of programs characterised by a large number of common features. The idea is to incorporate application semantics into the compilation process [32] by applying a range commonality analysis techniques. Due to the costs associated with compiler design and implementation, which are often disproportionally high to the size of typical DSLs, interpretation methods are often preferred. In his thesis [139], Thibault details a method by which a domain-specific abstract machine is obtained to serve as a basis for interpreter generation and later optimisation by means of partial evaluation. The emphasis there is on the automatic generation of efficient interpreters for DSLs and the method has been evaluated on languages for video device drivers [138] and active networking [137].

1.3.1.2 Methodologies of DSL design

These attempt to place DSL design and implementation into the wider context of domain engineering. Methods of domain engineering address the issue of identifying or creating reusable assets within a problem domain following thorough analysis of commonalities and variability among specific problem instances [7]. In this respect, requirements for a domain-specific language may be seen as a possible product of domain engineering.

The SDA (standing for Software Design Automation) methodology, proposed by Hook and Widen [145], takes the results of domain engineering as its starting point in guiding language design and implementation. It relies on building an essentially mathematical model of the domain, represented in a typed functional language. A candidate solution to the DSL design problem is subsequently validated with respect to the model, which also provides a formal semantics for the language. Thereafter, implementation follows techniques largely similar to those proposed by the advocates of embedded DSLs. Reliance on a formal semantics is advertised as one of SDA's merits. There is, however, no work known to us which uses this semantics as a foundation for the development of language-specific reasoning techniques. It also appears that the method is still in need of further evaluation, its current demonstration being in terms of a messaging system [143].

1.3.2 Visual (programming) languages

Owing to our interest in diagrammatic representations of software, our work naturally relates to developments in visual programming languages. To date, theoretical efforts in the field have focused mainly on the formulation, application and classification of grammatical techniques for graph specification and parsing (recognition). Relevant surveys are provided by Marriott and Meyer [95, 96, 97]. As Erwig remarks [33], it is mainly the *concrete* aspects of visual syntax that this research addresses. Grammatical formalisms are therefore deemed too cumbersome to serve as a basis for semantic definitions. Alternatively to regarding a visual language as the set of graphs producible by a grammar, one may attempt to obtain it as an algebra. More specifically, as the free algebra generated by applying a finite number of graph-based operations to a set of primitive visual programs.

Yet, the application of algebraic techniques in the description of visual languages (as opposed to the study of graph grammars and transformations) has been remarkably scanty. A notable exception is [142], where a Visual Object Definition Language is introduced, based on the principles of algebraic specification. Whereas the authors depart in advocacy of application-specific visual languages, their algebra itself is intended to be general (in the sense that a variety of visual notations could be defined in it). The view suggested in this thesis is that the choice of a particular algebra should be specific to the notation at hand. Instead of fixing any seemingly generic choice, attention should be shifted towards the collective meta-theoretic properties of notation-specific algebras and their relation to semantics.

1.3.3 Summary

On the whole, recent research on domain-specific languages has focused almost exclusively on textual languages and domains within mainstream computing (e.g. web computing [18] and networking [137]). In such domains, language specificity manifests itself primarily as "special [textual] syntax, operations and types" [77]. We wish to argue that, in general, the understanding of DSLs requires a context which is wider than simply "syntax" (or, even, semantics in the limiting sense that the term is applied to the semantically-driven construction of interpreters using techniques borrowed from general-purpose languages). Whereas it is widely agreed that a good DSL must capture the native notation and semantics of its application domain [17, 70], the range of languages studied thus far has led to a particularly narrow perception of both "notation" and "semantics". By emphasising industrial (engineering) domains and diagrammatic representations, our work aspires to contribute a genuinely new perspective.

1.4 Outline by chapter

The structure of this thesis can be conceptually divided into two parts:

- Part One is concerned with semantic and reasoning issues arising from the use of diagrams in domain-specific languages.

  - Chapter 2 introduces the domain of embedded control and IEC 1131-3, a class of industrial languages which are used to both motivate and illustrate our approach.
  - Chapter 3 formulates and demonstrates an approach to the design diagrammatic DSLs. The approach emphasises compositionality in the semantics of the language so as to allow the development of a simple proof system for inferring properties which are essential in the domain.
  - Chapter 4 deals with DSLs in which programs may be equivalently represented textually or diagrammatically. It establishes a rigorous consistency criterion regarding the two representations and details its application on the data-flow subset of IEC 1131-3.

- Part Two introduces an approach to defining the abstract syntax of diagrams in a way which both captures pragmatic aspects of their use and provides a basis for formal reasoning.
  - Chapter 5 introduces the mathematical preliminaries required in this part, namely the category of graphs and graph-homomorphisms and a variety of monoidal categories (herein referred to as *tensor categories*).
  - Chapter 6 motivates the need to include pragmatic aspects of diagrams in formalisations and demonstrates how a tensor category of metagraphs captures the abstract syntax and layout of control-flow diagrams in IEC 1131-3. Use of the formalism is subsequently illustrated in formulating inference rules for reasoning directly on the structure of diagrams.
  - Chapter 7 justifies the choice of metagraphs as specifications for IEC 1131-3 control diagrams and provides formal evidence that their category has the kind of algebraic structure required in Chapter 6.

1.5 Background requirements and conventions

A substantial amount of exposition in this thesis is mathematical in nature. The reader is assumed to possess a working knowledge of set theory and first-order (predicate) logic[1]. Some familiarity with universal algebra [99] will also be useful, so we provide a brief and simplified review of the basic concepts.

[1] Three texts which the author recommends as particularly gentle, pleasing-to-read introductions to sets and logic are [44, 133, 59]

1.5.1 Algebras, signatures and terms

A multi-sorted *algebra* is a collection $\mathcal{A}$ of sets, called the *carriers*, together with functions of the form $f : A_1 \times \ldots \times A_n \to A_m$ (where $A_k \in \mathcal{A}$), called *operations*, and possibly some distinguished elements c_i called *constants*.

A *signature* Σ is a collection of symbols partitioned into: a set S_Σ of *sort symbols*, a set K_Σ of *constant symbols*, and a set F_Σ of *function symbols*. Each constant symbol c is associated with a sort s, this association being written $c\colon s$. Each function symbol is associated with a string $s_1 \cdots s_{m+1}$ of sorts (i.e. an element of S_Σ^+), called the *arity* of the symbol, and one writes $f : s_1 \cdots s_m \to s_{m+1}$ to denote this.

An algebra for a signature Σ is a collection of sets A_s, one for each sort s in Σ, together with a distinguished element $a_c \in A_s$ for each constant symbol $c\colon s$ in Σ and a function $F_f : A_{s_1} \times \ldots \times A_{s_m} \to A_{s_{m+1}}$ for each function symbol f in Σ of arity $f : s_1 \ldots s_m \to s_{m+1}$.

Given any signature Σ and a set X whose elements are regarded as "variables", one may construct an algebra of terms (i.e. expressions) over Σ and X, denoted $\mathbb{T}(\Sigma, X)$. Each variable $x \in X$ is associated with a sort s, written $x\colon s$. All terms are formed as follows:

- c is "a term of sort s" for every constant symbol $c \in K_\Sigma$ such that $c\colon s$.
- x is "a term of sort s" for every variable $x \in X$ such that $x\colon s$.
- Given terms $t_1, \ldots, t_n$ of respective sorts $s_1, \ldots, s_n$ and a function symbol $f : s_1 \cdots s_n \to s_k$, then $f(t_1, \ldots, t_n)$ is a term of sort s_k. It is customary to use infix notation for binary function symbols and write "$t_1\, f\, t_2$" instead of "$f(t_1, t_2)$".

Each carrier of $\mathbb{T}(\Sigma, X)$ collects together all terms of the same sort. Each term of the form "c" is a constant (i.e. distinguished element). To recognise $\mathbb{T}(\Sigma, X)$ as an algebra, think of each term "$f(t_1, \ldots, t_n)$" as resulting from applying a function (corresponding to the symbol f) to the terms $t_1, \ldots, t_n$.

An *equation* over a term algebra $\mathbb{T}(\Sigma, X)$ is a pair $\langle t, t' \rangle$ of terms (of the same sort), conventionally written as $t = t'$. If $t = t'$ can be established by means of equational reasoning using a set E of equations as axioms, one writes $E \vdash t = t'$. This establishes an equivalence relation on $\mathbb{T}(\Sigma, X)$: two terms (of the same sort) are equivalent if they can be proved equal using E. The algebra $\mathbb{T}(\Sigma, X)/E$ in which each carrier collects all the equivalence classes of terms of the same sort is called the *quotient* of $\mathbb{T}(\Sigma, X)$ by E and has a very special property. Every other algebra which corresponds to the same signature Σ and in which equality is precisely captured by the axioms E is "essentially the same" (isomorphic) to the quotient algebra $\mathbb{T}(\Sigma, X)/E$.

1.5.2 Conventions

Definitions, propositions and examples are numbered with reference to the chapter in which they occur first. Thus, for example, "Definition 5.4" refers to the fourth definition in Chapter 5.

The proofs of propositions and lemmas are given in the text (as opposed to an appendix). Some of the longer proofs, especially in Chapters 4 and 7, may be skipped at first reading.

Chapter 2

The Domain of Embedded Control

This chapter introduces the domain of embedded control and its associated systems. This commonly occurring class of systems spans many different industries (e.g. automotive, process control, ASIC design, mobile telephony) and is a very common component of critical systems. The reason for our interest in embedded control is twofold:

- One of the first, principled and highly convincing arguments for the need to design and employ domain-specific languages originated in this domain [13].
- Most languages specific to the domain are naturally diagrammatic or admit both diagrammatic and textual representations of programs in order to cater for the needs of human users and system tools alike.

2.1 Reactive, embedded and Real-Time systems

Following a tradition established by Harel and Pnueli [61] (also [90], pages 3–5), the universe of computer programs is conceptually partitioned into those which are *transformational* and those which are *reactive*.

The purpose of a transformational program is to produce a set of outputs by applying a finite amount of computation to a given set of inputs. Thus, at least conceptually, changes in the inputs can only be accommodated in between program invocations. Examples in this class include compilers and programs for numeric computation (equation solvers, integrators etc.).

By contrast, a reactive program is one that maintains an ongoing interaction with its environment; during which it responds to a dynamically changing stream of inputs by dynamically producing an appropriate stream of outputs. Conventional examples of such programs include operating systems and graphical user interfaces. The most widespread application of reactive programs, however, is in the control of mechanical, chemical and other processes (e.g. engines and reactor plants). In such control applications, the reactions of the program aim to counteract changes detected in the state of the environment (the process being monitored), thereby controlling the environment's evolving behaviour.

Additionally, as it is commonly the case in control applications, the reactions of a reactive program may be constrained by specific temporal deadlines. Therefore, the correctness of such *real-time programs* not only depends on the functional aspects of their operation (logical correctness) but also, and critically so, on the timeliness of their behaviour (temporal correctness). In practice, timing constraints often mandate that specialised hardware is dedicated to the execution of real-time programs and that such hardware is, in turn, tightly coupled with the environment to ensure minimum latency in detecting and effecting change. In such cases, one speaks of *reactive systems* (that is, combinations of reactive software and specialised hardware) which are *embedded* in their environment. Most real-time control systems are embedded in this sense.

2.1.1 Models of reactive software

Mathematical functions provide an adequate, and also natural, abstraction of (the behaviour of) transformational programs. In particular, this view forms the basis of the *denotational* approach to the semantics of conventional programming languages [121]. In this framework, reference to time is made only when issues of termination or complexity are discussed. Even then, either a qualitative account of time passage suffices [65], or the semantic model is explicitly augmented with complexity measures [53].

On the other hand, the modelling of reactive programs is typically *operational* in style, relying on *transition systems* to provide a mathematical abstraction of their behaviour [90]. A transition system is, in essence, an automaton: the states represent discrete configurations of the program's state and each transition represents an action by the program to transform one state into another.

In order to deal with real-time software, the introduction of an explicit, quantitative account of time passage becomes inevitable. In this direction, transition systems have been extended in various ways. The two most popular approaches consist in the specification of temporal bounds (minimal and maximal delays) on transitions to obtain *timed transition systems* [66, 67], and the introduction of "clocks" (i.e. variables recording the passage of time) to obtain *timed automata* [1, 2].

In spite of recent progress, the current mathematical theory of real-time software is in many ways unsatisfactory and remains the subject of much ongoing research. Hoogeboom and Halang [69] discuss why computing lacks a stable notion of time by reviewing the philosophical, physical, mathematical, technological and social understandings of the concept. In particular, a significant problem regards the mathematical conception of time. Most commonly, time "values" are required to form a dense set, usually that being $\mathbb{R}$ (the set of real numbers). This, in turn, reflects a continuously varying environment which can unpredictably provoke events occurring at any point in time. Unfortunately, the complexity of reasoning over $\mathbb{R}$ seriously impedes the practical application of the theory to the verification of real-life systems. In response, techniques based on $\mathbb{Q}$ (the set of rational numbers) as an approximation (e.g. [58]) have been developed.

2.1.2 The synchronous approach to embedded control

In contrast to the general situation, the theory of embedded control software exhibits remarkable simplicity and has achieved a significant transfer of techniques into industrial use. Key to this success has been the observation that certain assumptions governing the application domain lead to simplifications in the design of specialised languages.

Among the earliest languages used in the domain were variants of general-purpose, imperative and concurrent languages such as Ada [13]. As a result of the asynchronous nature of process communication and the non-deterministic nature of concurrency in such languages, the behaviour and timing of control software became extremely hard to predict. In reaction to this unsatisfactory situation, a number of so-called *synchronous* languages were developed in the 1980's, the most prominent among which being Esterel [16, 14], Lustre [57, 20], Signal [38], Statecharts [62] and Argos [91]. The design of synchronous languages rests on the idealisation, called the *synchrony hypothesis* [10, 89], that each reaction of the system is instantaneous. Any reference to "real time" within synchronous programs may then be eliminated, resulting in the development of simple semantic models [12, 15, 92, 21].

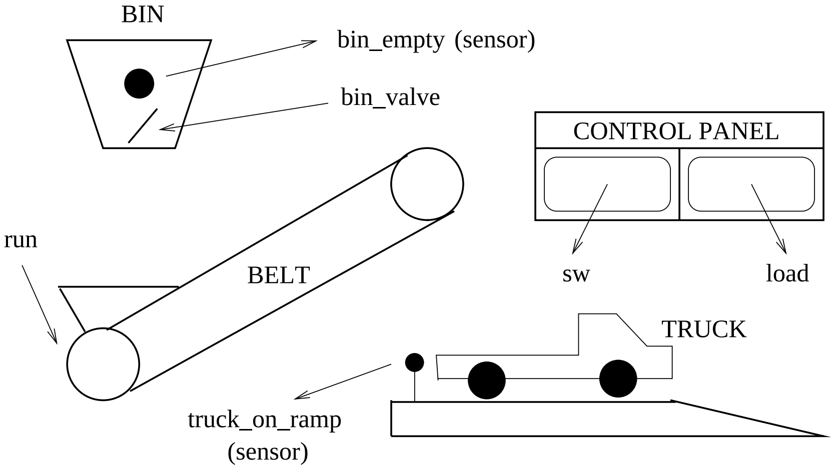


Figure 2.1: A gravel dump site. Labels having arrows pointing towards them represent inputs to the PLC, whereas labels with outgoing arrows represent PLC outputs.

2.2 Programmable logic controllers

Programmable logic controllers [135, 111] (PLCs) are a class of embedded systems developed specifically for applications in industrial process control. As the name implies, a PLC is programmed to compute a particular, usually mode-dependent and time-varying, relation between a set of controlled outputs and a set of process-related inputs.

Figure 2.1 (adapted from an example in [23], Annex F) presents the setup of a typical PLC application. Here the task is to transfer a measured amount of gravel from a bin onto a track under human supervision. Human control is exercised through a panel of two on/off switches, the status of which is represented as boolean variables sw and load in the PLC system. The other two inputs to the controller come from two sensors represented as boolean variables bin_empty and truck_on_ramp respectively. Based on these inputs the controller computes boolean outputs run and bin_valve to control the motion of the conveyor belt and the valve at the bottom of the bin.

Control theory models processes as vectors of continuously varying quantities, called signals. The inputs to a PLC are discritised (i.e. quantised through sampling) representations of these signals. In the preceding example, for instance, the input bin_empty is a boolean quantisation of the quantity of gravel in the bin, which may assume any real value between 0 and the capacity of the bin. Discretisation is performed by an array of devices called *sensors*. Similarly, the outputs from the PLC are effected upon the controlled process through an array of *actuators*. These, in turn, drive a variety of electromechanical devices, such as motors, pumps or relays. The embedding of a PLC in the context of a plant is illustrated in Figure 2.2.

Driven by a digital clock, a PLC repeats a simple three-step cycle consisting of:

1. reading the values of the sensors (inputs) on the occurrence of each clock pulse;

2. computing *before* the next occurrence of a clock pulse, and by means of executing instructions in software, new values for the outputs based on the current inputs; and

3. releasing the new output values to the actuators at the next occurrence of a clock pulse.

In practice, sophisticated latching mechanisms are employed to ensure that the inputs presented to the PLC are stable when they are read at the beginning of each cycle; correspondingly, the outputs are also latched before they are applied at the beginning of the next cycle. Diagrammatically, this sequence of events is illustrated in Figure 2.3.

A critical assumption in the application domain is that the clock of the PLC is much faster (sometimes by orders of magnitude) than the rate at which the signals representing the environment change. This justifies one in applying the synchrony hypothesis, with important ramifications to the design of PLC software.

In general, the control program executed by a PLC consists of multiple, communicating units which operate concurrently (as suggested in Figure 2.2). Since all computation and communication internal to the PLC must take less than a clock cycle, synchrony enables the view that internal communication delays are zero and that all internal components compute in constant time. This view presents the control engineer with a greatly simplified, yet adequate, model of computation.

2.3 The IEC 1131-3 standard

In 1993, PLCs became the subject of IEC International Standard 1131. Part 3 of this standard [23, 86], defines a suite of four domain-specific languages which have become known collectively as IEC 1131-3. Programs in these languages may be represented either diagrammatically or textually and the two forms of

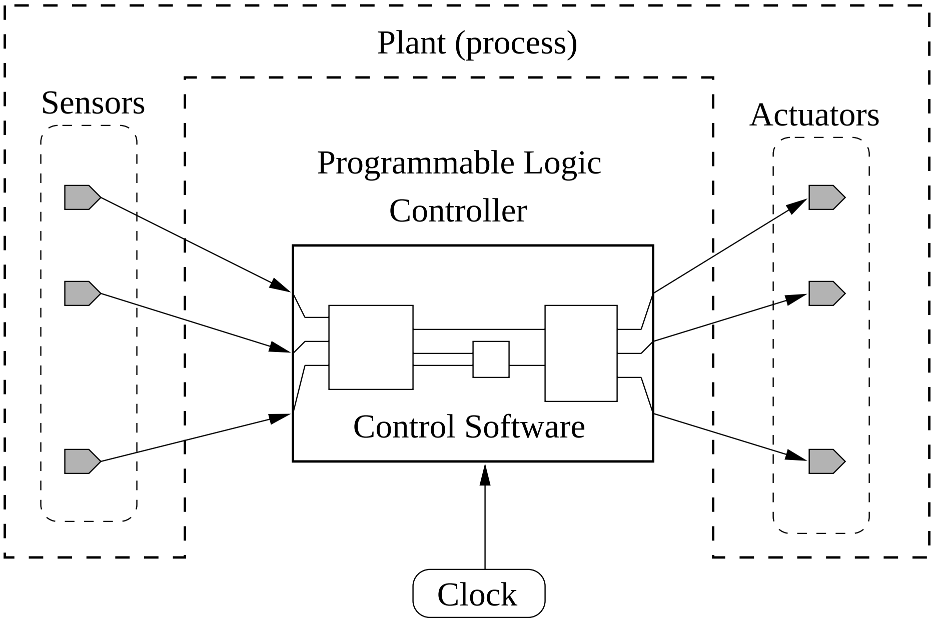


Figure 2.2: Embedding of a PLC system.

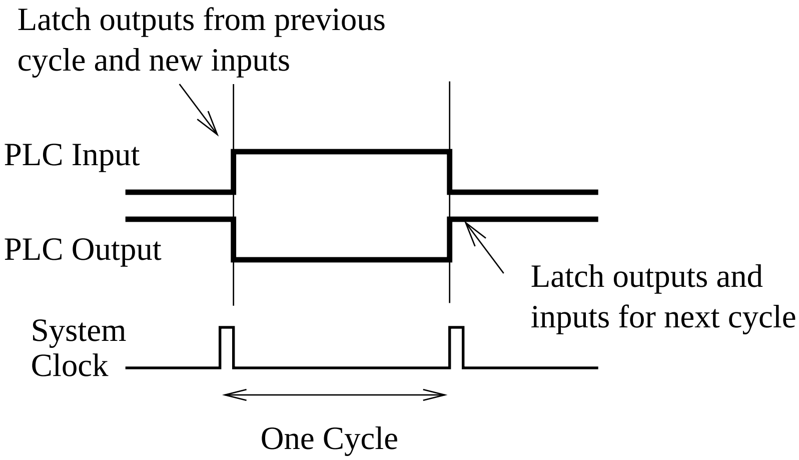

Latch outputs from previous cycle and new inputs
PLC Input
PLC Output
Latch outputs and inputs for next cycle
System Clock
One Cycle

Figure 2.3: Three-step PLC cycle.

representation are declared equivalent by the standard. The conceptual elements of PLC programs reflect the two main aspects of control programming:

1. *Function Blocks* specify data dependencies between individual outputs and inputs of the system.

2. *Sequential Function Charts* (SFCs) specify how the overall mode of the system (i.e. the overall input-output relation) changes over time in response to internal and external events.

2.3.1 Function Blocks and their diagrams

Function blocks are the basic units of PLC programs and encourage the designer to use predefined components which are then composed to form control programs. Figure 2.4 shows an example of the graphical declaration of a function block in IEC 1131-3 notation. The block uses two named instances "DB_ON" and "DB_OFF" of a predefined timer block ("TMR") and an instance "DB_FF" of a predefined "SR" flip-flop block to debounce a binary input "IN". The outputs of the block are "OUT" and "ET_OFF". The inputs and outputs of each block constitute its external interface and, by appeal to the diagrammatic representation, we shall call them the *ports* of the block.

Function block diagrams are appealing to the user because they resemble typical circuit schematics developed by control engineers for describing hard-wired controllers. For the purposes of machine interpretation, however, function blocks are also given a textual representation in a language called Structured Text (ST). Figure 2.5 shows the textual declaration corresponding to our example function block. Labels (and types) for inputs and outputs are introduced by the keywords VAR_IN and VAR_OUT, whereas VAR introduces labels for instances of predefined blocks. The body of the block consists of assignments to the output labels with references to the input labels and the outputs of block instances (in the form INSTANCE.OUTPUT).

In essence, function blocks are simple, synchronous data-flow [11] programs specifying the outputs as functions of the inputs. In general, the function computed may be dependent on the history of its own computation. History dependencies are introduced by means of *feedback loops* linking outputs to inputs as illustrated in Figure 2.6. In this example, the new value for A in each cycle is the value of Q computed in the previous cycle. Explicit initialisation of A is required to provide a value in the first cycle. In ST, loops are implemented with the aid of "local variables" whose values persist between invocations:

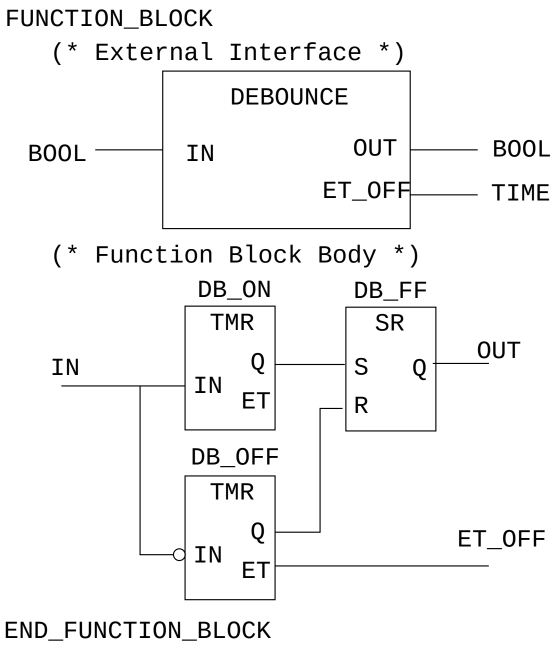


Figure 2.4: Example of graphical function block declaration.

```
FUNCTION_BLOCK DEBOUNCE
(** External Interface **)
  VAR_INPUT
    IN    : BOOL;
  END_VAR
  VAR_OUTPUT
    OUT   : INT;
    ET_OFF : TIME;
  END_VAR
  VAR
    DB_ON : TMR;
    DB_OFF : TMR;
    DB_FF : SR;
  END_VAR
(** Body **)
  DB_ON( IN := IN ); DB_OFF( IN := NOT IN );
  DB_FF( S := DB_ON.Q, R := DB_OFF.Q );
  OUT := DB_FF.Q; ET_OFF := DB_OFF.ET;
END_FUNCTION_BLOCK
```

Figure 2.5: Function block DEBOUNCE in ST.

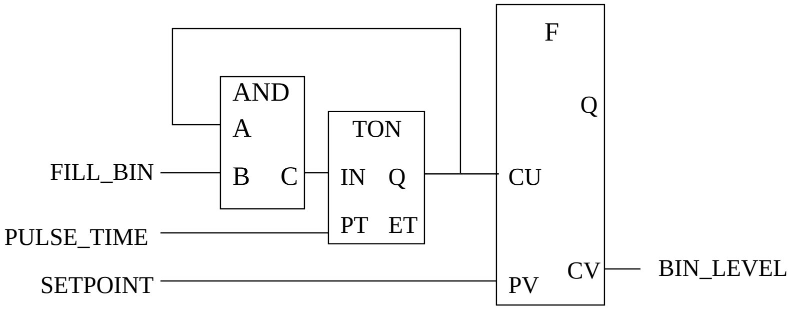


Figure 2.6: Block diagram containing a loop from the output Q of TON to the input A of AND.

```
FUNCTION BLOCK ...
  VAR_INPUT ...
  VAR_OUTPUT ...
  VAR OLDQ : TYPE := INIT_VAL; ...
BEGIN
  AND( A := OLDQ, B := ... );
  TON( ... );
  F( CU := TON.Q, ... );
  OLDQ := TON.Q
END
```

2.3.2 Sequential Function Charts

The diagrammatic representation of SFCs is illustrated in Figure 2.7 on a possible solution to the gravel transferring problem of Figure 2.1. The graphical notation employed is that of a net comprising linked elements of following kinds:

- rectangular boxes, called *steps*;
- thick horizontal lines, called *transitions*; and
- a variety of branching elements.

In such nets, no elements of the same kind may be linked directly.

Each step is labelled with an identifier, e.g. "DUMP" in Figure 2.7, and is optionally associated with an *action* which is typically displayed within an oblong attached box. Actions are either function block programs or simple truth-value assignments to boolean variables. Transitions, on the other hand, are labelled by logical conditions (i.e. boolean expressions)[1] and are said to be *cleared* when

[1] For convenience and consistency with the rest of the paper we shall write transition conditions using traditional logical notation instead of the standard IEC 1131-3 syntax. In particular, tt and ff are used here as abbreviations for the logical constants "true" and "false" respectively.

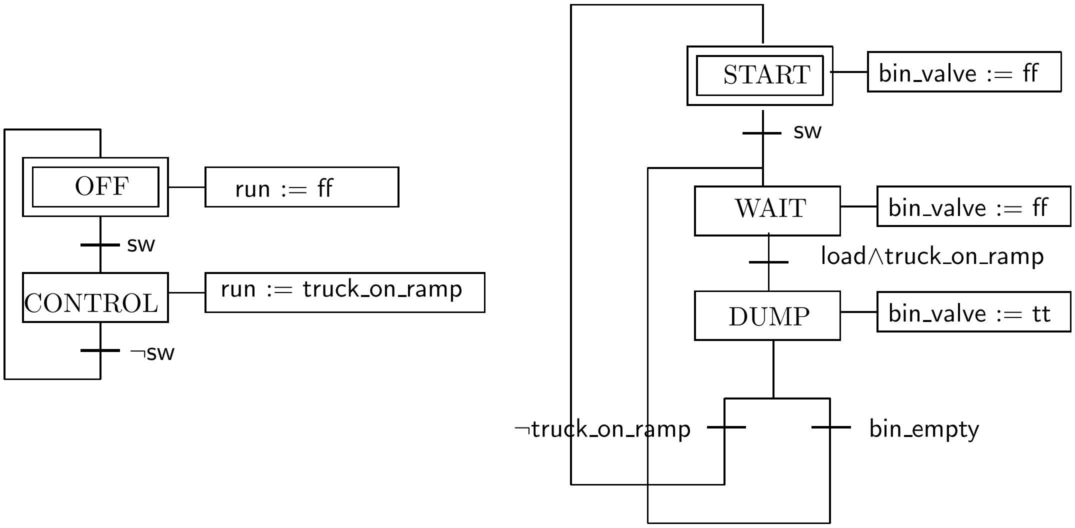


Figure 2.7: Example SFC diagram.

their corresponding condition evaluates to 'true'. Finally, a number of steps can be designated as *initial*; these are distinguished by a double border.

SFCs exhibit a rich control-flow behaviour (dynamics). At any given time, each step can be either active or inactive and the set of all active steps defines the current mode of the system. In each mode, only the actions of all active steps are invoked. A step remains active until one of its successor transitions becomes cleared, thereby causing the steps targeted by the links emanating from that transition to become active in the next cycle.

The construction of SFCs is structured in a way that parts of the diagram form certain kinds of processes. The rules of processes construction are called *rules of evolution* and are illustrated in Figure 2.8.

- Construction (a) is a simple linear sequence. Control passes from `S3` to `S4` upon clearance of `t`.

- Construction (b) is a *sequence divergence*, or "*fork*". Control flows from `S5` to `S6` if `t1` is cleared, or from `S5` to `S6` if `t2` is cleared. The situation generalises to more than two branches.

- Construction (c) is a *sequence convergence*, or "*join*", in which control evolves from `S7` to `S10` if `S7` is active and `t1` is cleared, or from `S9` to `S10` if `S9` is active and `t2` is cleared. The situation generalises to more than two branches.

- Construction (d) allows the concurrent activation of more than one processes. In the given example, control passes from `S3` to both `S4` and `S5`

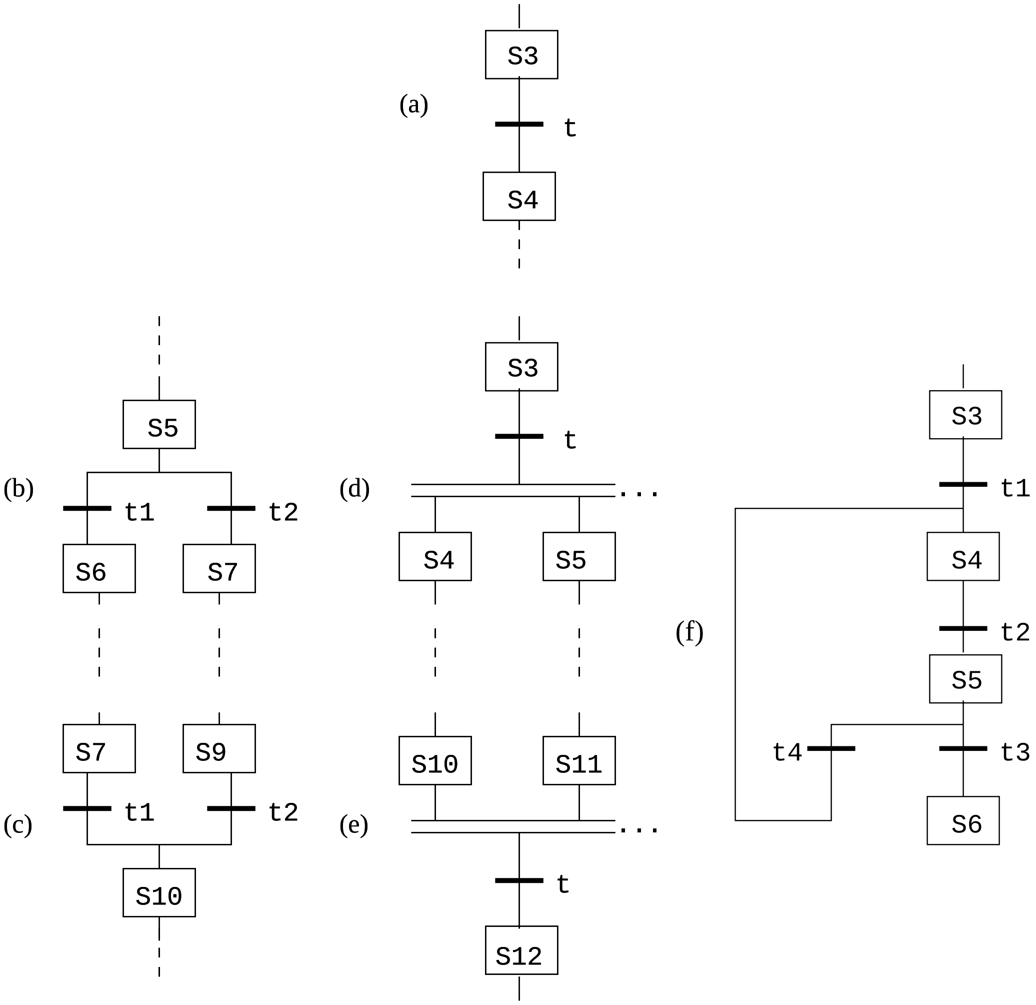


Figure 2.8: SFC evolution rules. Adapted from [23].

upon clearance of `t`. Following their simultaneous activation, the processes beginning at `S4`, `S5` evolve independently.

- The construct of case (e) concludes a number of concurrent processes. In the given example control evolves from `S10` and `S11` to `S12`, if *both* `S10` and `S11` are active and `t` is cleared.

- Finally, case (f) is a special case of sequence divergence in which one of the branches loops back to an earlier step.

SFCs are also provided with a textual representation, the details of which will not concern us here beyond saying that it comprises three lists of objects (steps, transitions and actions), and that each object contains references to the objects that are connected to it.

2.4 Discussion

The IEC standard provides a formal (grammatical) definition for only the textual variants of PLC languages. The syntax of diagrams is illustrated informally by way of example. On the whole, this situation demonstrates existing methods of visual language definition to have made only limited impact on most practical applications. On the other hand, it leaves implementations of the standard with considerable freedom in translating diagrams into textual forms. The possibility of unsound, or unexpected translations is obvious. A considerable part of this thesis is concerned with addressing this problem and, more generally, with the formal definition of diagrams for domain-specific languages.

Examples are also the preferred method for defining the semantics of PLC languages in the IEC 1131-3 standard. For many critical applications of PLCs, this form of definition is clearly inadequate [56, 72] and, as argued in [141], it is partly ambiguous and confusing. Formal semantic definitions have been attempted separately for a subset of function blocks [36] and Grafcet [93, 22], the French standard from which SFCs evolved. A unified, operational semantic framework for both paradigms was developed in the author's Masters thesis [140]. (Based on that earlier account, the semantics of SFCs presented in Chapter 3 is, however, denotational in style, entirely new and specific to the present thesis.)

Chapter 3

Design for Proof

We propose that the domain of a domain-specific language can be characterised by:

1. the class of environments in which systems developed in the language are expected to operate; and

2. the class of properties which such systems are expected to possess.

The design of DSLs should therefore include the development of a proof system that eases the task of proving the properties in the class identified for the anticipated operating environments.

We develop these ideas in the context of industrial computing systems by presenting a semantics and proof system for a language based on IEC SFCs. Of particular significance in this work is the use of a diagrammatic representation and the development of a proof system for a class of invariance properties that requires only local knowledge of the structure of diagrams.

3.1 Introduction

In general, the particular domain for which a DSL is developed will determine characteristically:

1. The kind of models that programs in the language denote (i.e. the semantics); and

2. The kind of representations that programs admit (i.e. the form(s) in which programs are presented to the user).

For instance, signal processing programs in Signal [38] have an intuitive interpretation as digital, synchronous circuits. On the other hand the same language admits, in addition to textual syntax, a graphical representation of programs resembling the block diagrams used by the signal processing community.

Our view is that application domains can also be characterised by some class of properties that users want *easily* proven of programs in the domain. The degree to which the presence of such important properties is made evident in the representation of programs should therefore be a major quality requirement in many domains. Programs written in languages which are “designed for proof” should require little or even no extra effort to exhibit the requisite properties. To this end, the present chapter advocates the contributing role that formal models of reasoning can play in the design and evaluation of DSLs.

Languages for industrial critical systems provide excellent material upon which the above ideas can be applied. Firstly, the importance of certain classes of properties is apparent: industrial software is often subjected to strict certification based on various safety and other quality requirements [56]. Secondly, software representation in industrial systems is often diagrammatic and derived from existing design practices.

Despite being generally appealing, many of the diagrammatic notations used for programming purposes are often described as “confusing”. Most commonly, these are notations which lack a compositional semantics in the sense of permitting spatially dispersed and seemingly unrelated parts in a diagram to engage in subtle interaction. Confusion then arises when such interactions are not explicitly represented in the diagram by means of a spatial relation, e.g. a visual “link” between the interacting parts. As a result, reasoning must involve arguments about the global structure of potentially large diagrams.

In this chapter, the problem of inferring properties based on diagrams is addressed in a domain-specific context. In particular, it is shown how properties may be proved locally by:

1. Eliminating features in the notation which introduce non-compositional behaviour; and

2. Exploiting domain-specific assumptions to reduce the complexity of the reasoning tasks for which basic support is sought. More specifically, the complexity aspect to be minimised in our case is the dependency on computational history of the formulae in which properties are expressed.

3.2 Design for Proof

Domain-specific programming languages are used to represent the artifact that is the object of a design process. Designers are concerned to provide evidence that their design is fit for its purpose. This suggests that in designing DSLs one should attempt to make the task of providing such evidence as easy as possible.

Viewing this formally, evidence is a collection of properties one wants to hold of the program together with a proof that the program satisfies the properties. This suggests a characterisation of the application domain by:

1. The class of properties that will be useful in providing evidence that a system written in the language is fit for its purpose.

2. A characterisation of the environments in which the system will operate.

The designer of a DSL should then:

1. Develop the semantics of the language so that programs can be developed that satisfy all (or most) properties in the class of interest.

2. Develop a sound proof system for proving that programs satisfy properties (one may also require that the system is complete in some suitable sense).

Important further considerations in the design of a DSL are the pragmatics of the language:

1. That the representation of programs is well-matched to the domain.

2. That proofs are easy to carry out, perhaps there is a semi-automated search procedure that succeeds most of the time.

3. That good diagnostic information is available in the event of failure to show some property.

3.3 Compositionality

One widely argued advantage of designing domain-specific languages is that programs in such languages should be easily analysable and that their properties should be easily inferable. One necessary — but not in any measure sufficient — condition for this is that the proof systems associated with these language should be simple and tractable. In this and the next section we clarify what we mean by "simple proof systems" and examine the repercussions of this requirement to the design of the languages themselves.

3.3.1 Compositional inference systems

Proof systems for domain-specific languages should be as specific as the languages themselves. In particular, they should be "representation-aware" in the sense that the form of logical discourse they support relates closely to the domain-specific representation of programs.

By this we mean that proof obligations should be formulated directly in terms of the user-level representation of programs. Each such obligation takes the form

$$P \vdash \phi \ ,$$

(read "P entails ϕ") where P is some term corresponding to (the abstract syntax of) a program in the language and ϕ is a logical formula expressing some property of interest. In particular, P should bear an evident relation to the concrete representation of the program it stands for; be that representation textual or diagrammatic.

On the other hand, the requirement of *simplicity* relates both to the form of inference rules and the complexity of the logic in which reasoning is conducted. Proof systems for DSLs should be "representation-directed" also in the sense that they are *compositional.* That is, the rules for composite programs take the form:

$$\frac{P_1 \models \phi_1 \quad \cdots \quad P_n \models \phi_n}{\mathcal{C}(P_1, \ldots, P_n) \models \Phi_{\mathcal{C}}(\phi_1, \ldots, \phi_n)} \ ,$$

where $\mathcal{C}(P_1, \ldots, P_n)$ stands for some structural combination of terms $P_1, \ldots, P_n$ and $\Phi_{\mathcal{C}}$ is an n-ary function on formulae[1]. In words, every way of combining components induces a corresponding way for combining the properties of each individual component to yield a property of the composite program.

3.3.2 Compositionality in semantics

The *soundness* of inference rules is argued based on a mathematical model of the semantics for the language at hand. Thus, the ability to formulate compositional proof rules relies upon the existence of a semantics for the language which is similarly compositional. More precisely, a semantic mapping h is compositional if for every composite program $\mathcal{C}(P_1, \ldots, P_n)$ one has

$$h(\mathcal{C}(P_1, \ldots, P_n)) = M_{\mathcal{C}}(h(P_1), \ldots, h(P_n))$$

for an appropriate function $M_{\mathcal{C}}$, the definition of which does not depend on the specific programs $P_1, \ldots, P_n$.

[1] I.e. the result of $\Phi_{\mathcal{C}}(\phi_1, \ldots, \phi_n)$ is a new formula composed out of sub-formulae of the given $\phi_1, \ldots, \phi_n$

Inability to produce compositional rules for a language is therefore regarded as evidence of a mismatch between the language's syntax and semantics. In other words, certain possibly important aspects of a program's behaviour are not adequately reflected in its representation.

3.3.3 The choice of logic

Compositionality is a necessary (but not sufficient) condition for simplicity of inference. Even in entirely compositional proof systems, further complexity can arise from the particular choice of logic over which formulae range. Some logics, such as the modal mu-calculus [131], are known to contain compact formulae whose models are disproportionally intricate compared to the formula's structure. This impacts heavily on the complexity of inference systems which are complete over the entire logic, i.e. they are capable of deriving $P \vdash \phi$ whenever $M(P) \models \phi$ where ϕ is an arbitrary formula and $M(P)$ a suitable model of P.

Domain-specificity offers the opportunity of not only developing domain specific logics, but also of restricting attention to just fragments of logical theories. Such fragments should be expressive enough to cover most properties of interest in the domain and, hopefully, simple enough to permit elementary inference. In embedded control, for instance, most required properties follow particular patterns and are predicated upon known properties of earlier designs. Identifying the nature and form of such patterns can then be seen as a natural objective of domain analysis methods.

3.4 Homomorphicity, systematicity

The point of having a representation in the first place is that it bears some degree of resemblance to whatever it is being represented. Owing to this similarity, a representation reflects some aspects of the represented entity which are deemed relevant in a particular context, while, possibly, ignoring others. There is, however, a multitude of notions of "similarity" and varying degrees of constrains that they impose. A framework for making such notions precise has recently been put forward by Gurr [52, 46].

This framework is illustrated in Figure 3.1 and consists in abstracting both the representation and the represented in terms of mathematical entities which Gurr called α-worlds (standing for "abstract worlds").

Definition 3.1. An α-world W consists of a set X of objects and a set R of relations over X. ■

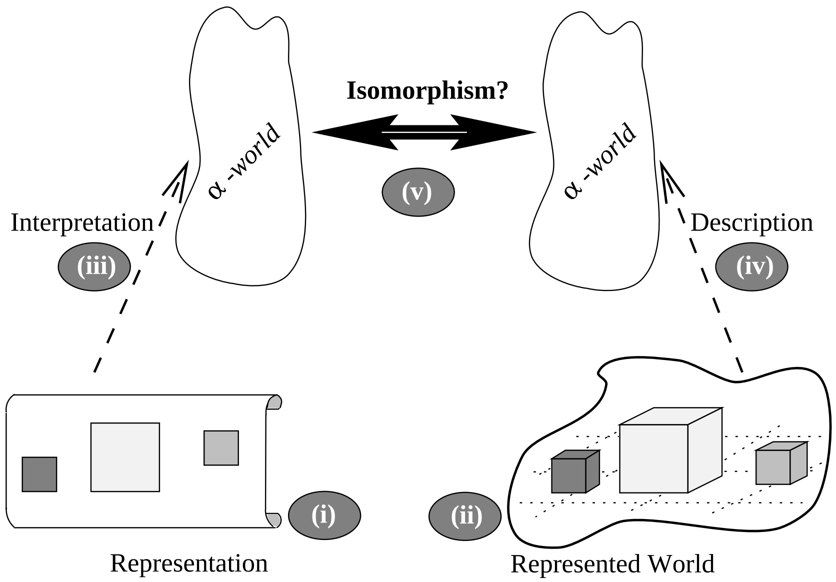


Figure 3.1: Capturing the relationship between represented and representing worlds: a representation (**i**) (a diagram, say) and the situation it represents (**ii**) are described by two α-worlds. The first corresponds to some reader's interpretation (**iii**) of the representation while the second is a description (**iv**) of the relevant parts of the represented worlds. To say that the representation is isomorphic (homomorphic) to the represented is to say that the mapping (**v**) is an isomorphism (homomorphism). (Adapted from [52].)

This allows one to define notions of mapping between α-worlds in a way which clearly parallels definitions in algebra and therefore benefits from a commensurate degree of rigour:

Definition 3.2. A (general) mapping f from α-world W to α-world W' is a pair $f_X : X \to X'$, $f_R : R \to R'$ of functions (although one frequently blurs notation and denotes both f_X and f_R as f). A mapping h from W to W' is a *homomorphism* if for every relation which holds between objects in W, the corresponding relation in W' holds between the corresponding objects. Thus, a homomorphism h from W to W' is a mapping such that for all $r \in R$ and $x_1, \dots, x_n \in X$

$$(x_1, \dots, x_n) \in r \implies (h(x_1), \dots, h(x_n)) \in h(r) \ .$$

■

Consider now a situation such as in Figure 3.1, where the two α-worlds W, W' adequately capture everything which is deemed relevant in a representation and the represented. One then says that the representation is *homomorphic* to the represented if a homomorphism h exists from W to W'.

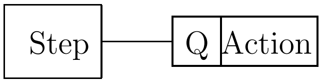
Let us now examine the situation in the opposite direction, assuming as before that W' satisfactorily captures every relation in the represented entity which we regard as important. What does the existence of a homomorphism h' from W' to W signify? In part, it tells us that every such important relation has a corresponding relation in the representation. Additionally, and most importantly, it tells us that every (first-order) logical statement about the relations in W' translates to a statement about W which holds if the original does in W'.

Following the terminology of Gurr [51], we say that the representation captured by W is *systematic* over the represented world W', or has the property of *systematicity*, if a homomorphism $h' : W' \to W$ exists. Systematicity therefore induces logical properties on the representing relations which "match" those of the represented. What makes many diagrammatic representations *effective* for reasoning purposes is that the induced properties are easily "read off" the diagram, and one is thereby assisted in concluding that corresponding relations must hold in the represented. In [51], Gurr expresses the same fact by saying that certain spatial relations holding in diagrams are "directly" semantically interpretable (considering the represented as providing the semantics of its representation). As we shall see, this view hints a subtle link between systematicity and compositionality.

3.5 Lack of systematicity in IEC SFCs

Representations which are systematic in the above sense are a key ingredient of our "design-for-proof" approach to domain-specific languages. In this section we take a critical look at one particular feature permitted by the definition of SFCs in the IEC standard, and expose how it contravenes the objective of facilitating reasoning.

The semantic model associated with SFCs is that of (labelled) Petri nets, a concept widely used and well understood in the domain [149], and the basic SFC notation is highly suggestive of this association. Unfortunately, the definition of SFCs in [23] abounds with extensions to the basic Petri net concepts that forcefully violate this analogy. One such extension permits certain actions to be "set active" by some step and continue to be invoked following the step's deactivation. Such actions will remain active either indefinitely or until they are explicitly "reset" by a step elsewhere in the diagram. This mechanism of action-step association, called called *action qualification*, is visualised by attaching an oblong box to a step as follows:

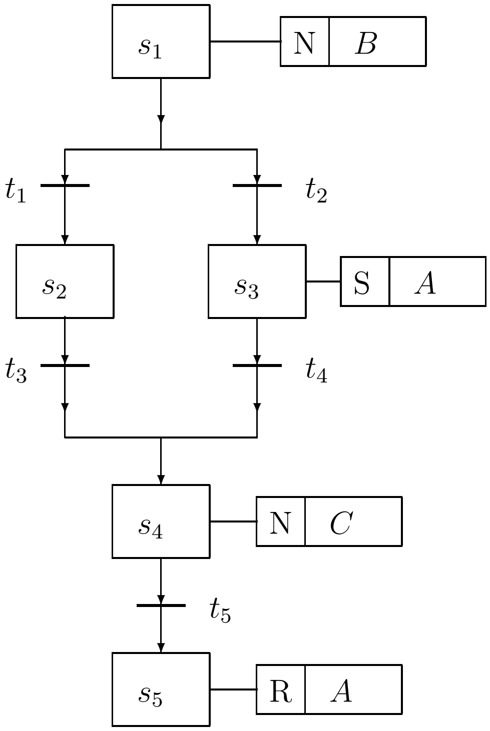


,

where Q is a "qualifier". Of the many qualifiers permitted, here we look at "N" (usually omitted and standing for "normal") and "S", "R" (standing for "set" and "reset").

Consider now the SFC diagram in Figure 3.2 which makes use of this feature. One possible α-world abstraction of this diagram captures the *static structure* (statics), i.e. what one sees in the diagram in terms of visual objects and how they are visually related (by means of links and associations). So the α-world in question has objects $s_1, \ldots, s_5, A, B, C, t_1, \ldots, t_5$, corresponding to the steps, actions and transitions. The relations in the α-world capture links and action-step associations. The relation L is such that $(x, y) \in L$ iff a directed link exists from x to y in the diagram (e.g. $(t_1, s_2) \in L$, but $(t_5, s_4) \notin L$). For each qualifier and action a there is a binary relation Q such that $(x, a) \in Q$ iff x is a step associated with a via Q. So, for example, $S = \{(s_3, A)\}$ and $N = \{(s_1, B), (s_4, C)\}$. Let us call this α-world D as it captures the statics of the diagram.

The labelled-net semantics of our example diagram is given (also diagrammatically!) in Figure 3.3. Each place in the net is labelled with zero or more actions and the α-world P associated with the net has objects for the places, transitions and labels (actions). Its relations are F, corresponding to the "flow" of the net,



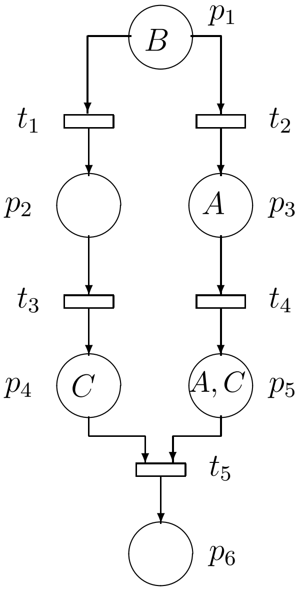
Objects:

$$s_1, \dots, s_5$$
$$t_1, \dots, t_5$$
$$A, B, C$$

Relations:

$$\begin{aligned} L &= \{(s_1,t_1),(s_1,t_2),(t_1,s_2),\\ &\quad (t_2,s_3),\dots\} \\ N &= \{(s_1,B),(s_4,C)\} \\ S &= \{(s_3,A)\} \\ R &= \{(s_5,A)\} \end{aligned}$$

Figure 3.2: SFC diagram and its α-world D.



Objects:

$$p_1, \dots, p_6$$
$$t_1, \dots, t_5$$
$$A, B, C$$

Relations:

$$\begin{aligned} F &= \{(p_1,t_1),(p_1,t_2),(t_1,p_2),\\ &\quad (t_2,p_3),\dots,(p_4,t_5),\dots\} \\ M &= \{(p_1,B),(p_3,A),\\ &\quad (p_4,C),(p_5,A),(p_5,C)\} \end{aligned}$$

Figure 3.3: Corresponding net and its α-world P.

and a relation M such that $(p, a) \in M$ iff place p is labelled with action a.

We now proceed to evaluate the degree of correspondence between the diagram and its semantics (i.e. between the statics and the implied dynamics). For this purpose we form an α-world U which provides (partial) information about a user's interpretation of the dynamics as inferred from the statics of the SFC diagram, and we seek a homomorphism from P to U.

The α-world U has the same objects and relation L as D but only one additional relation $G = N \cup S$. In particular, G contains exactly those action-step associations which are explicitly guaranteed in the diagram, and thus hold in all semantic interpretations which respect the meaning of qualifiers. Thus, for example, $(s_4, A) \notin G$ as this association depends on the history of the computation leading to s_4.

One now observes that there can be no homomorphism from P to U, as every candidate should map both p_4 and p_5 to s_4 and $(p_5, A) \in M$ whereas $(s_4, A) \notin G$. *We are forced to conclude that an important semantic relation, that of which actions are invoked in each mode of the system, is not systematically visualised.* This introduces complications in reasoning and suggests that the introduction of the "S" and "R" qualifiers poorly integrates with the core notation.

Finally, observe that the way in which qualification violates systematicity also incurs non-compositionality. In general, there is no way of dividing the net of Figure 3.3 into parts, represent each part (systematically) as an SFC diagram and combine the resulting diagrams in a similar manner to form our example SFC.

By elementary analysis, we have thus shown how some questionably convenient features of IEC SFCs can introduce a seriously dangerous mismatch between a user's intuitive interpretation of the graphical representation and its actual semantics. In the presence of such features, knowledge of the global structure of the SFC may be required before overall behaviour can be inferred from the behaviours of the currently active steps. Such knowledge may be extremely hard to establish accurately about large diagrams.

3.6 A rational re-design of SFCs

In this section, and in accordance to our proposed methodology emphasising ease of proof, we offer an alternative design of SFCs based on:

1. A characterisation of the environments in which SFCs operate;

2. The interpretation of a class of properties expressible in first-order logic as simple invariants of input-output relations;

3. The use of only a basic diagrammatic representation which is free of features responsible for non-local behaviour; and finally,

4. The development of a simple, intuitive and compositional semantics which is straightforwardly reflected in the chosen representation. This leads to a similarly simple and compositional proof system.

Simplicity is therefore given priority over mathematical novelty, unless of course the latter is required to achieve the former. The intention, however, is not to impose simplicity *artificially* on PLC languages. Instead, we wish to argue that conceptual clarity is naturally present in the primitive notions of step, action and transition and that much of the complexity associated with SFCs in IEC 1131-3 is extraneous and largely attributable to the standardisation process. The latter, being an attempt to reconcile a number of diverse approaches to PLC programming, often resorted to the conglomeration of many heterogeneous features instead of identifying a small, well-understood core [141].

Despite being elementary in nature, our framework suffices to model a large class of SFCs in IEC 1131-3, either directly or through a simple translation scheme. It can be further extended to include other features of IEC 1131-3, e.g. step activity flags, which are not in conflict with our interpretation of the graphical notation.

3.6.1 Characterising the domain

When designing a DSL, it is important to account for specific assumptions which are characteristic of the application domain. In our case, the operating environment of a PLC is assumed to produce signal changes at a rate which is much slower than the speed of the PLC itself. This both suggests and justifies a particular design of PLC languages in which interaction with the environment is perceived as being synchronous.

Characterising the environment through synchrony allows for the simple modelling of PLC programs: The behaviour of SFCs can be linearly but adequately captured as *traces*, i.e. finite or (countably) infinite sequences of states:

$$t : \; s_0 \; \ldots \; s_{n-1} \; \ldots \; .$$

Each adjacent pair (s_i, s_{i+1}) of states in a trace constitutes a single cycle (cf. Figure 2.3).

Another characterisation of the application domain is in terms of the classes of properties which users would like to easily prove of programs. Control applications are usually characterised by at least the following two basic requirements:

- a *responsiveness* requirement that the system will always react within a finite amount of time to a change of stimulus from the controlled environment;
- a *safety* requirement that the reaction is always the right one.

In PLCs the responsiveness issue has been resolved at the level of language design by requiring that the evaluation of all actions terminates within the time allowed for each cycle.

The safety aspect, on the other hand, lies entirely within the programmer's responsibility. To enhance dependability, established development practices require actions to be composed of standard function blocks drawn from verified libraries. The latter are usually specific to a particular company, supplying vendor or industrial sector [84, 55]. Nevertheless, behaviour which is custom to the application at hand is still the result of putting these standard actions together in SFCs. One is therefore interested in showing that a given SFC succeeds in maintaining some desired safety invariant throughout its computation. Most commonly, this will be a relation between only the *current* values of inputs and outputs (i.e. not dependent on computational history).

3.6.2 Choosing the logic

One suitable and sufficiently simple language for expressing such input-output relations is first-order logic augmented with "primed variables". Each formula A in the logic will be interpreted wrt. pairs (s, s') of states, where s and s' provide valuations for the unprimed and primed variables in A respectively. Taking (s, s') to be the current execution cycle, the primed variables in A will intuitively stand for the new outputs and the new inputs at the end of the cycle. Finally, given a trace t as above, one wishes to determine whether A is an *invariant* of t:

Definition 3.3. $t \models A$ iff $(t[j], t[j+1]) \models A$ for all $0 \leq j <\mid t \mid -1$. The *length* $\mid t \mid$ of t is defined to be $n+1$ for $t = s_0 \ldots s_n$ and $\mid t \mid \overset{\text{def}}{=} \omega$, the first infinite ordinal, for every infinite trace t. ■

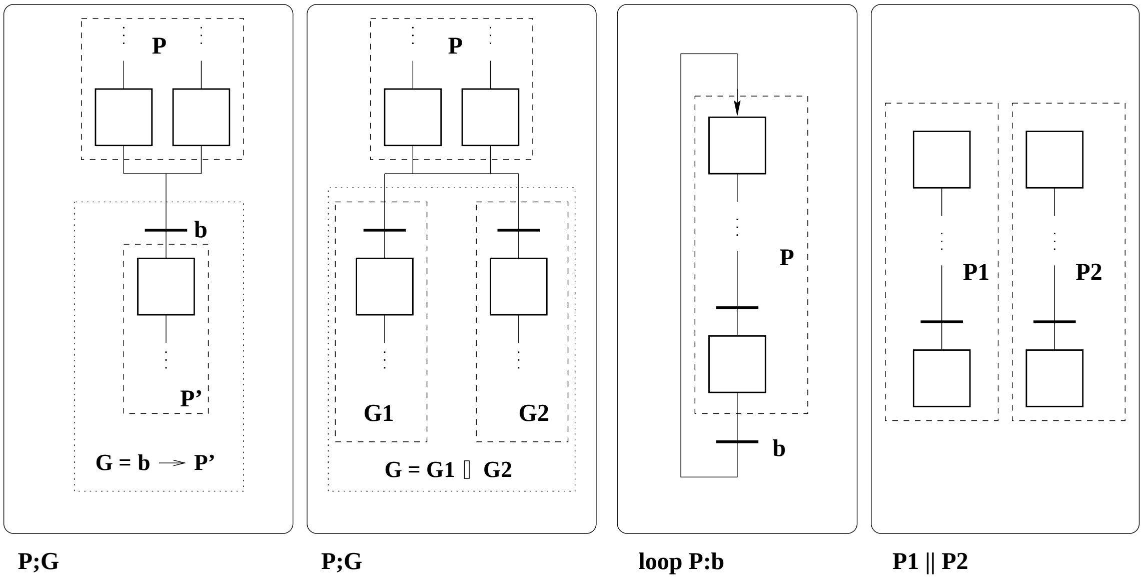


Figure 3.4: Syntax-diagram correspondence.

3.7 An abstract syntax for SFC diagrams

A textual syntax for the expression of SFC diagrams is first introduced as a basis for defining a formal semantics:

$$\begin{aligned} P &::= \langle a \rangle \mid P\,;G \mid P_1 \,\|\, P_2 \mid \text{loop}\ P\colon b \\ G &::= b \rightarrow P \mid G_1 \talloblong G_2 \end{aligned}$$

Here, the phrase types are *programs*, ranged over by P, and *guarded programs*, ranged over by G. On the other hand, a is an action and b stands for the condition for performing a transition. $\langle a \rangle$ stands for a single step with associated action a, $P\,;G$ is sequential composition via the transitions in G and $P_1 \,\|\, P_2$ is parallel composition. Finally, loop $P\colon b$ is a loop with body P and transition b along the link to the top of the loop. The correspondence of this syntax to the diagrammatic notation is illustrated in figure 3.4. In particular, the composite $G_1 \talloblong G_2$ is to be interpreted as non-deterministic choice between the branches G_1 and G_2.

3.7.1 Program Gravel translated

As an example of using the syntax just introduced a translation of the diagram of figure 2.7 is now described. The entire diagram is just $P_1 \,\|\, P_2$ where

- $P_1 \equiv \text{loop}\ (\langle a_1 \rangle\,; \mathsf{sw} \rightarrow \langle a_2 \rangle)\colon \neg\mathsf{sw}$ is the translation of the SFC on the left with actions $a_1 \equiv \mathsf{run} := \mathsf{ff}$ and $a_2 \equiv \mathsf{run} := \mathsf{truck\_on\_ramp}$;

- $P_2 \equiv \text{loop } (\langle a_3 \rangle \,;\mathsf{sw} \rightarrow P_3)\colon \mathsf{bin\_empty}$ corresponds to the SFC on the right with $P_3 \equiv \text{loop } (\langle a_3 \rangle \,; (\mathsf{load} \wedge \mathsf{truck\_on\_ramp} \rightarrow \langle a_4 \rangle))\colon \neg\mathsf{truck\_on\_ramp}$ being the translation of the inner loop and $a_3 \equiv \mathsf{bin\_valve} := \mathsf{ff}$, $a_4 \equiv \mathsf{bin\_valve} := \mathsf{tt}$.

3.8 A compositional semantics of SFC diagrams

3.8.1 Semantics of actions

Action computation obeys a well-known synchronous dataflow paradigm, the semantics and properties of which have been studied extensively, e.g. [11, 12]. In denotational style, the behaviour of signals over time is typically modelled as finite or (countably) infinite sequences of values, called *streams.* The i-th element in a stream is the value of the represented signal at the i-th point on a discrete time line corresponding to the ticks of the system's clock. Let A^ω denote the set of streams with elements from set A. A (deterministic) dataflow agent of n input signals and m output signals is then modelled as a *monotone* and *continuous* [146] function

$$f : (I_1 \times \ldots \times I_n)^\omega \rightarrow (O_1 \times \ldots \times O_m)^\omega$$

where I_i is the set of values which the i-th input may assume over time and O_j is the set of values for the j-th output. Continuity of f means that its behaviour on infinite streams is entirely determined by its behaviour on finite ones.

Since actions in SFCs have named inputs and outputs, we assume a set V of variable names (or signal names) and a set $\mathcal{V}$ of basic values, and define $\mathcal{S} \overset{\text{def}}{=} V \rightarrow \mathcal{V}$ to the set of *states*; each state being an association of all variable names with values. Invoking an action in a given state produces new values for the outputs of the action. Thus, we denote an action a by a monotone and continuous function

$$[\![a]\!] :\ \mathcal{S}^\omega \rightarrow (\gamma(a) \rightarrow \mathcal{V})^\omega \ \ ,$$

where $\gamma(a)$ is the set of outputs of a (i.e. the variables that a may modify). Furthermore, each $[\![a]\!]$ is required to be length-preserving[2].

Then, the set of infinite traces resulting from the repeated invocation of a in an environment which does not modify the variables in $\gamma(a)$ is:

$$\mathrm{Tr}_\infty(a) \overset{\text{def}}{=} \{t \in \mathcal{S}^\omega \mid |\,t\,| = \omega \text{ and } \forall j \in \mathbb{N}\,.\ t[j+1] \restriction \gamma(a) = ([\![a]\!]t)[j]\} \ \ ,$$

[2] I.e., for all $t \in \mathcal{S}^\omega$, $|\ ([\![a]\!]\, t)\ | = |\ t\ |$. Unlike programs in [12], function blocks are always single-clocked.

where $t[j]$ and $f \restriction X$ denote the j-th state of trace t and the restriction of map f to X, respectively.

In practice, the environment in which a operates will control only a known, finite set E of variables such that $E \cap \gamma(a) = \emptyset$. When no ambiguity arises, we shall sacrifice precision in favour of brevity and refer to the set E as just *the environment*. To account for the execution of actions in SFCs, one must also consider all possible finite (but non-empty) traces of actions. Thus, given action a and $E \subseteq V$, define the set of possible traces of a in environment E as: $\mathrm{Tr}^{E}_{\geq 2}(a) \overset{\text{def}}{=} \emptyset$ if $E \cap \gamma(a) \neq \emptyset$ and

$$\mathrm{Tr}^{E}_{\geq 2}(a) \overset{\text{def}}{=} \left\{ \; t \in \mathcal{S}^{\omega} \;\middle|\; \begin{array}{l} t \sqsubseteq_{\geq 2} t_{\infty} \text{ for some } t_{\infty} \in \mathrm{Tr}_{\infty}(a) \text{ such that} \\ \forall x \in V \,.\, x \notin E \cup \gamma(a) \implies t_{\infty} \models x' = x \end{array} \right\}$$

otherwise, where $t \sqsubseteq_{\geq 2} t'$ iff t is a prefix of t' (denoted $t \sqsubseteq t'$) and $|t| \geq 2$.

3.8.2 Semantics of SFC programs

The meaning of an SFC program P is now identified with the set $[\![P]\!]^{E}_{\Psi}$ of traces it is capable of producing in environment E subject to some *exit condition* Ψ. The role of Ψ is to account for the effect of any successor transitions in the context of sequential composition. Similarly, a set $[\![G]\!]^{E}_{\Psi}$ of traces is also defined for each guarded program G, environment E and transition condition Ψ.

3.8.2.1 Step

When P is a single step $\langle a \rangle$, $[\![\langle a \rangle]\!]^{E}_{\Psi}$ consists of all traces t resulting from *at least one* invocation of a such that:

- Ψ never holds following the initial invocation (in which case t is infinite); or

- t is a finite trace in $\mathrm{Tr}^{E}_{\geq 2}(a)$ and the last state in t satisfies Ψ.

Adopting the convention of writing $s \models A$ as a shorthand for $(s, s) \models A$ whenever A contains no primed variables, one formally has:

$$\begin{aligned} [\![\langle a \rangle]\!]^{E}_{\Psi} \;&= \{ t \in \mathrm{Tr}^{E}_{\geq 2}(a) \;|\; |t| = \omega \text{ and } \forall j \geq 1 .\, t[j] \models \neg\Psi \} \,\cup \\ &\left\{ \; t \in \mathrm{Tr}^{E}_{\geq 2}(a) \;\middle|\; \begin{array}{l} |t| = n < \omega,\ \forall 1 \leq j \leq n-2 \,.\, t[j] \models \neg\Psi \\ \text{and } t[n-1] \models \Psi \end{array} \right\} . \end{aligned}$$

3.8.2.2 Sequential Composition

A trace of $P \,; G$ is defined as the concatenation of a trace of P and a trace of G, thus corresponding *exactly* to the intuitive interpretation of sequence induced by the diagrammatic representation.

Definition 3.4. Given two traces $t_1, t_2 \in \mathcal{S}^\omega$, their concatenation $t_1 \circ t_2$ is only defined when:

- either $|t_1| = \omega$, in which case $t_1 \circ t_2 = t_1$;
- or $t_1 = s_0 \ldots s_n$ and $t_2 = s_n\, s_{n+1} \ldots$, in which case:

$$t_1 \circ t_2 = s_0 \ldots s_n\, s_{n+1} \ldots . \quad \blacksquare$$

Given any two sets $T_1, T_2 \subseteq \mathcal{S}^\omega$ of traces, let $T_1 \oplus T_2$ be the set of all possible traces resulting by legitimately concatenating traces in T_1 with traces in T_2:

$$T_1 \oplus T_2 \overset{\text{def}}{=} \{t_1 \circ t_2 \mid t_1 \in T_1,\, t_2 \in T_2\} \enspace .$$

In particular, $T \oplus \emptyset = \emptyset \oplus T = \emptyset$ for all $T \subseteq \mathcal{S}^\omega$. The semantics of sequence is now formally defined as:

$$[\![P\,; G]\!]^E_\Psi = [\![P]\!]^E_{\text{cond}(G)} \oplus [\![G]\!]^E_\Psi \enspace ,$$

where cond(G) is the assertion associated with the conditions on the initial transitions in G, defined as: cond$(b \rightarrow P) \overset{\text{def}}{=} b$ and cond$(G_1 [\!] G_2) \overset{\text{def}}{=}$ cond$(G_1) \vee$ cond(G_2).

3.8.2.3 Parallel Composition

The semantics of parallel composition in 1131-3 is, at least in our opinion, inadequately defined and obscurely presented. In particular, the relevant clauses in the standard prescribe no method of concurrency realisation, either through interleaving or otherwise. Furthermore, reasoning about concurrent SFCs is greatly complicated by the presence of many *ad-hoc* features (e.g. delayed action execution) which, although questionably convenient for reducing the size of program representation, are by no means essential. Instead, they introduce behavioural complications which are extremely hard to deduce from the diagrammatic representation, particularly in large SFCs.

This unfortunate situation is to be sharply contrasted with function blocks which obey a simple dataflow model. Let, for instance, a_1 and a_2 be two function blocks with inputs I_1 and I_2 and outputs O_1 and O_2 respectively. These two blocks may be composed in parallel provided that they share no outputs ($O_1 \cap O_2 = \emptyset$). The inputs and the outputs of the composite block are then $I_1 \cup I_2$ and $O_1 \cup O_2$ respectively. This model is inherently concurrent and interaction between the two components is only permitted through their inputs and outputs at the boundaries between successive invocation cycles. Consequently, this form of composition is also well-behaved in the following sense: if A_1 and A_2 are formulae characterising the current invocation cycle of a_1 and a_2 respectively, then the current invocation of $a_1 \,\|\, a_2$ is characterised by $A_1 \wedge A_2$.

It is only reasonable to expect this property of 'static' block composition to be preserved when actions are 'dynamically' composed in parallel under the control of SFCs. This requirement, which is not guaranteed in 1131-3, forces us to restrict parallel composition only to programs P_1 and P_2 which are *disjoint* in the following sense:

Definition 3.5. Let $\gamma(P) \overset{\text{def}}{=} \bigcup_{a\in\mathrm{Act}(P)} \gamma(a)$ where $\mathrm{Act}(P)$ is the set of all actions in P. Programs P_1 and P_2 are called *disjoint* if $\gamma(P_1) \cap \gamma(P_2) = \emptyset$. ■

In words, P_1 and P_2 are disjoint if none of the actions in P_1 shares an output with an action of P_2 and vice-versa. No other restriction is placed on either the inputs of any action or the transition conditions, in either P_1 or P_2. Now, the above requirement on the behaviour of $P_1 \,\|\, P_2$ can be formalised as:

$$\llbracket P_1 \,\|\, P_2 \rrbracket^E_\Psi = \llbracket P_1 \rrbracket^{E_1}_\Psi \cap \llbracket P_2 \rrbracket^{E_2}_\Psi \quad (P_1, P_2 \text{ disjoint}) \ ,$$

where $E_1 = E \cup \gamma(P_2)$ and $E_2 = E \cup \gamma(P_1)$. Observe that when $P_1 = \langle a_1 \rangle$ and $P_2 = \langle a_2 \rangle$, $P_1 \,\|\, P_2$ behaves exactly as the parallel composition of the two function blocks a_1 and a_2.

3.8.2.4 Loop

The repetitive behaviour of loops is captured by

$$\llbracket \text{loop } P\colon b \rrbracket^E_\Psi = \mathrm{fix}(\lambda \mathrm{T}\,.\,\llbracket \mathrm{P} \rrbracket^{\mathrm{E}}_\Psi \cup (\llbracket \mathrm{P} \rrbracket^{\mathrm{E}}_{\mathrm{b}} \oplus \mathrm{T})) \ , \tag{3.1}$$

where fix(F) stands for the least fixed point[3] of function F.

It is worth pointing out that equation (3.1) abstracts from the behaviour of loops in 1131-3 by allowing a non-deterministic choice of which condition (b or Ψ) is attended at the end of each loop iteration. An interpretation which conforms to the standard can be obtained by simply replacing $\llbracket P \rrbracket^E_b$ in (3.1) with

$$\llbracket P \rrbracket^E_{b\vee\Psi} \setminus \{t \mid \ |t| = n < \omega \ \wedge \ t[n-1] \models \Psi\} \ .$$

Equation (3.1) is therefore adequate for showing invariants of SFCs in 1131-3 and is preferred here for its simplicity.

[3] In (3.1), the least fixed point exists because $\oplus$, and hence $\lambda T\,.\,\llbracket P \rrbracket^E_\Psi \cup (\llbracket P \rrbracket^E_b \oplus T)$, is clearly monotonic.

3.8.2.5 Guarded Programs

Finally, guarded programs receive the following simple interpretations:

$$\begin{aligned} [\![b \to P]\!]^E_\Psi &= \{t \in [\![P]\!]^E_\Psi \mid t[0] \models b\} \ , \\ [\![G_1 [\!] G_2]\!]^E_\Psi &= [\![G_1]\!]^E_\Psi \cup [\![G_2]\!]^E_\Psi \ . \end{aligned}$$

In particular, the modelling of $G_1 [\!] G_2$ as non-deterministic choice presents a generalisation of *sequence divergence* (fork) in IEC 1131-3 where left-to-right precedence of transitions is usually assumed.

3.9 The proof system

Our design effort now yields straightforward proof rules for showing safety properties of the kind identified in section 3.6.1. Safety is semantically characterised in terms of relations:

Definition 3.6. $a \models_E A$ iff $t \models A$ for all $t \in \mathrm{Tr}^E_{\geq 2}(a)$. ■

In words, each relation $a \models_E A$ expresses that every E-computation of action a exhibits invariant A.

Definition 3.7.

$$\{\Phi\}\, P\, \{\Psi\} \models_E A$$

iff $t \models A$ for all traces $t \in [\![P]\!]^E_\Psi$ such that $t[0] \models \Phi$. (Φ and Ψ are assumed to contain no primed variables.) ■

In words, $\{\Phi\}\, P\, \{\Psi\} \models_E A$ holds iff formula A is an invariant of every trace representing a computation of program P started in a state satisfying Φ and under exit condition Ψ. A relation of the form $\{\Phi\}\, G\, \{\Psi\} \models_E A$ is defined similarly.

Figure 3.5 lists the inference rules for the entire language. Each judgement of the form $\{\Phi\}\, P\, \{\Psi\} \vdash_E A$ is interpreted as asserting that $\{\Phi\}\, P\, \{\Psi\} \models_E A$ holds. Judgements of the form $\{\Phi\}\, G\, \{\Psi\} \vdash_E A$ receive a similar interpretation.

Assuming the soundness of the rules with respect to the semantic interpretations, a derivation of $\{\Theta\}\, P\, \{\mathsf{ff}\} \vdash_E A$ will therefore be a proof that A is an invariant of the complete program P in environment E and under initial condition Θ. A proof of the soundness theorem below can be found in the Appendix.

Proposition 3.1. *If $\{\Phi\}\, P\, \{\Psi\} \vdash_E A$ is derivable using rules (3.2)–(3.9) then $\{\Phi\}\, P\, \{\Psi\} \models_E A$.*

$$\frac{}{\{\Phi\}\,\langle a\rangle\,\{\Psi\} \vdash_E A}\; a \models_E \Phi \vee \neg\Psi \implies A \tag{3.2}$$

$$\frac{\{\Phi\}\,P\,\{\mathrm{cond}(G)\} \vdash_E A \quad \{\mathrm{cond}(G)\}\,G\,\{\Psi\} \vdash_E A}{\{\Phi\}\,P\,;G\,\{\Psi\} \vdash_E A} \tag{3.3}$$

$$\frac{\{\Phi\}\,P_1\,\{\Psi\} \vdash_{E\cup\gamma(P_2)} A_1 \quad \{\Phi\}\,P_2\,\{\Psi\} \vdash_{E\cup\gamma(P_1)} A_2}{\{\Phi\}\,P_1 \,\|\, P_2\,\{\Psi\} \vdash_E A_1 \wedge A_2} \tag{3.4}$$

$$\frac{\{\Phi \vee b\}\,P\,\{\Psi \vee b\} \vdash_E A}{\{\Phi\}\,\mathrm{loop}\ P\colon b\,\{\Psi\} \vdash_E A} \tag{3.5}$$

$$\frac{}{\{\Phi\}\,P\,\{\Psi\} \vdash_E A}\; \Phi \implies \mathsf{ff} \tag{3.6}$$

$$\frac{\{\Phi\}\,P\,\{\Psi\} \vdash_E A'}{\{\Phi\}\,P\,\{\Psi\} \vdash_E A}\; A' \implies A \tag{3.7}$$

$$\frac{\{\Phi \wedge b\}\,P\,\{\Psi\} \vdash_E A}{\{\Phi\}\,b \rightarrow P\,\{\Psi\} \vdash_E A} \tag{3.8}$$

$$\frac{\{\Phi\}\,G_1\,\{\Psi\} \vdash_E A \quad \{\Phi\}\,G_2\,\{\Psi\} \vdash_E A}{\{\Phi\}\,G_1 \| G_2\,\{\Psi\} \vdash_E A} \tag{3.9}$$

Figure 3.5: Proof rules

Proof. See Section 3.12. ■

Our aim in designing the semantics of SFCs and the associated proof system was ease of proof, not mathematical novelty. In particular, two aspects of the proof system are worth noting:

1. *Support for a proof strategy based on localised arguments.* The correctness of any step S wrt. some property A is *locally* evident in the vicinity of S, that is in the action a of S and the immediately preceding and succeeding transition conditions. This is reflected in rule (3.2), which also highlights the role of the two transitions Φ and Ψ as summaries of the information about the behaviour of the environment while S is active. Localised reasoning was also made possible through the exploitation of domain-specific assumptions to reduce the complexity of properties to a level expressible in first order logic instead of, say, some temporal logic.

2. *Easy to reason about concurrency.* Compared to most corresponding rules in various proof systems for shared-variable concurrency, e.g. [132, 147], rule (3.4) looks unrealistically simple. However, this is only partly the result of disregarding the issue of completeness of the proof system. Pri-

marily, though, it is the result of observing a model of parallelism which is appropriate to the domain at hand. The parallel composition of SFCs was *naturally* derived from the simple notion of concurrency provided by the underlying circuit-like model which individual actions obey. Apart from maintaining consistency with the level of function blocks, this interpretation allows $P_1 \,\|\, P_2$ to be thought of in terms of logical conjunction, as suggested by rule (3.4). The definition of SFCs in the IEC standard fails to provide such a simple characterisation.

Unlike many similar proof systems, the one presented here is not geared towards program analysis or the synthesis of programs from specifications. As such, it was not meant to support any particular approach to program development (e.g. top-down or bottom-up). Instead, the purpose of the rules and of the soundness theorem is to provide formal justification for simple reasoning (both formal and informal) about the safety of SFCs based on their diagrammatic representation. As a result of taking this particular point of view, completeness of the rules was of lesser importance to our development.

Remark. *The condition* $a \models_E \Phi \vee \neg\Psi \implies A$ *in rule (3.2) is rather strong, but preferred here for reasons of clarity. Alternatively, one could have defined a safety relation*

$$a \models^{E}_{\Phi,\Psi} A$$

as $t \models \phi$ *for all* $t \in [\![\langle a \rangle]\!]^{\Psi}_{E}$ *such that* $t[0] \models \Phi$*. That is,* A *is invariant of every* E*-computation of* a *which starts in a state satisfying* Φ *and in which* Ψ *holds invariantly thereafter. Under this definition, rule (3.2) would be:*

$$\frac{}{\{\Phi\}\,\langle a \rangle\,\{\Psi\} \vdash_E A}\; a \models^{E}_{\Phi,\neg\Psi} A\ .$$

3.10 Verifying Gravel

The inference rules of the previous section are now illustrated by proving a simple invariant property of program Gravel: namely that whenever the truck leaves the ramp, the program will close the bin valve and stop the belt at the next cycle.

So formally, let $A \equiv \neg\mathsf{truck\_on\_ramp} \implies \neg\mathsf{bin\_valve}' \wedge \neg\mathsf{run}'$. (For each boolean variable x we abbreviate formulae $(\mathsf{x} = \mathsf{tt})$ and $(\mathsf{x}' = \mathsf{tt})$ as x and x' respectively.) We prove that $\{\mathsf{tt}\}\, P_1 \,\|\, P_2\, \{\mathsf{ff}\} \models_E A$, where P_1 and P_2 are exactly as given in Section 3.7 and $E = \{\mathsf{truck\_on\_ramp}, \mathsf{bin\_empty}, \mathsf{sw}, \mathsf{load}\}$.

First, rules (3.4) and (3.7) suggest that one seeks properties A_1 and A_2 which together imply the desired property A such that A_1 is an invariant of P_1 and A_2 an invariant of P_2. Two suitable such properties are $A_1 \equiv \neg\mathsf{truck\_on\_ramp} \implies \neg\mathsf{run}'$ and $A_2 \equiv \neg\mathsf{truck\_on\_ramp} \implies \neg\mathsf{bin\_valve}'$. The root of the proof tree is thus:

$$\dfrac{\dfrac{\Delta_1 \quad \Delta_2}{\{\mathsf{tt}\}\, P_1 \,\|\, P_2 \,\{\mathsf{ff}\} \vdash_E A_1 \wedge A_2} \quad A_1 \wedge A_2 \implies A}{\{\mathsf{tt}\}\, P_1 \,\|\, P_2 \,\{\mathsf{ff}\} \vdash_E A}\ (3.4)$$

where Δ_1 is the sub-derivation of $\{\mathsf{tt}\}\, P_1 \,\{\mathsf{ff}\} \vdash_{E_1} A_1$, $E_1 = E \cup \{\mathsf{bin\_valve}\}$, and Δ_2 is the the proof of $\{\mathsf{tt}\}\, P_2 \,\{\mathsf{ff}\} \vdash_{E_2} A_2$, $E_2 = E \cup \{\mathsf{run}\}$. Let us first consider Δ_1 (in which environment information has been suppressed):

$$\dfrac{\dfrac{\dfrac{}{\{\mathsf{tt}\}\,\langle a_1\rangle\,\{\mathsf{sw}\} \vdash A_1}\ (3.2) \quad \dfrac{\dfrac{}{\{\mathsf{sw} \wedge \mathsf{sw}\}\,\langle a_2\rangle\,\{\neg\mathsf{sw}\} \vdash A_1}\ (3.2)}{\{\mathsf{sw}\}\,\mathsf{sw} \to \langle a_2\rangle\,\{\neg\mathsf{sw}\} \vdash A_1}\ (3.8)}{\{\mathsf{tt} \vee \neg\mathsf{sw}\}\,(\langle a_1\rangle\,;\mathsf{sw} \to \langle a_2\rangle)\,\{\mathsf{ff} \vee \neg\mathsf{sw}\} \vdash A_1}\ (3.3)}{\{\mathsf{tt}\}\,\mathrm{loop}\,(\langle a_1\rangle\,;\mathsf{sw} \to \langle a_2\rangle)\colon \neg\mathsf{sw}\,\{\mathsf{ff}\} \vdash A_1}\ (3.5)$$

Now $a_1 \models (\mathsf{tt} \vee \neg\mathsf{sw}) \implies A_1$, which is equivalent to $a_1 \models \neg\mathsf{run}' \vee \mathsf{truck\_on\_ramp}$, holds clearly as a_1 always sets run to ff. Similarly, $a_2 \models (\mathsf{sw} \vee \neg\neg\mathsf{sw}) \implies A_1$ is equivalent to $a_2 \models \neg\mathsf{sw} \vee (\neg\mathsf{truck\_on\_ramp} \implies \neg\mathsf{run}')$ and holds because every evaluation of $\mathsf{run} := \mathsf{truck\_on\_ramp}$ satisfies $\mathsf{run}' \implies \mathsf{truck\_on\_ramp}$.

Moving now to the proof that P_2 has A_2 let us consider tree Δ_2:

$$\dfrac{\dfrac{\dfrac{}{\{\mathsf{tt}\}\,\langle a_3\rangle\,\{\mathsf{sw}\} \vdash A_2}\ (3.2) \qquad \Delta_{21}}{\{\mathsf{tt} \vee \mathsf{bin\_empty}\}\,(\langle a_3\rangle\,;\mathsf{sw} \to P_3)\,\{\mathsf{ff} \vee \mathsf{bin\_empty}\} \vdash A_2}\ (3.3)}{\{\mathsf{tt}\}\,\mathrm{loop}\,(\langle a_3\rangle\,;\mathsf{sw} \to P_3)\colon \mathsf{bin\_empty}\,\{\mathsf{ff}\} \vdash A_2}\ (3.5)$$

Requirement $a_3 \models (\mathsf{bin\_empty} \vee \neg\mathsf{sw}) \implies A_2$ is equivalent to $a_3 \models \mathsf{truck\_on\_ramp} \vee \neg\mathsf{bin\_valve}'$ and therefore holds for reasons similar to those applied in the case of a_1. Δ_{21} considers the inner loop P_3 and is very similar to Δ_1.

3.11 Discussion

We have outlined an approach to the design of domain-specific languages. Our approach emphasises ease of proof for classes of properties associated with the application domain. Our understanding of proof in this context is based on a formal model of the language and its programs and on a set of sound inference rules. The essence of domain specificity lies in assumptions which are usually far stronger than those holding in general-purpose languages. This results in considerable uniformity across programs in the same language and domain owing to a large number of common characteristics. Domain specificity can therefore be exploited to focus a formal approach to language design on precisely those aspects, features and properties of the language for which good design support is needed most. It is in this way, we argue, that domain-specific languages lend themselves to semantic and logical models which are clear and tractable.

Our approach was illustrated by presenting a model for a core language corresponding to sequential function charts in IEC 1131-3. In this experiment, only a basic diagrammatic representation of programs was assumed which, however, reflects the essential concepts in applications. The model of the language was compositionally developed on the structure of diagrams and in a way corresponding to intuition. Motivated by ease-of-proof, our design exercise resulted in a set of straightforward but sound proof rules which validate reasoning about a narrowed but commonly occurring class of safety properties. Moreover, use of our rules requires only local knowledge about the representation of SFC programs as diagrams.

3.12 Proof of Proposition 3.1

3.12.1 Preliminaries

Sets of traces together with the usual subset ordering $\subseteq$ form a simple *complete partial order* (cpo) [146] with bottom element $\emptyset$. Let T range over sets of traces. $\_ \oplus \_$ is continuous in its right argument. Hence, function $F = \lambda T \,.\, [\![P]\!]^E_\Psi \cup ([\![P]\!]^E_b \oplus T)$ is continuous and, by a standard result of fixed-point theory,

$$[\![\mathrm{loop}\ P\colon b]\!]^E_\Psi = \bigcup_{n \in \mathbb{N}} T_n \qquad (3.10)$$

where $T_0 = \emptyset$ and $T_{n+1} = F(T_n)$ for all $n \geq 0$. The following lemmas are simple consequences of the definitions in Section 3.8

Lemma 3.1. *For all $t \in [\![P]\!]^E_\Psi$ or $t \in [\![G]\!]^E_\Psi$, $|\,t\,| \geq 2$. Moreover, if $|\,t\,| = n < \omega$ then $t[n-1] \models \Psi$.*

Informally, Lemma 3.1 states that any trace of P results from at least one execution of the initial step(s) in P and, moreover, the last state of any *finite* such trace satisfies the exit condition Ψ.

Lemma 3.2. *For all $t \in [\![G]\!]^E_\Psi$, $t[0] \models \mathrm{cond}(G)$.*

3.12.2 Main proof

By rule induction [146]. We consider each rule in turn:

Rule 3.2 Assume $a \models_E (\Phi \vee \neg\Psi) \implies A$. Fully expanded, this assumption reads:

$$\forall t \in \mathrm{Tr}^E_{\geq 2}(a).\ \forall\, 0 \leq j < \mid t \mid -1.\ (t[j], t[j+1]) \models (\Phi \vee \neg\Psi) \implies A$$

or, equivalently,

$$\forall t \in \mathrm{Tr}^E_{\geq 2}(a).\ \forall\, 0 \leq j < \mid t \mid -1.\ \begin{array}{l}(t[j], t[j+1]) \models \Phi \vee \neg\Psi \text{ implies} \\ (t[j], t[j+1]) \models A\ .\end{array} \tag{3.11}$$

Now consider $t \in [\![\langle a \rangle]\!]^E_\Psi$ such that $t[0] \models \Phi$. There are two sub-cases:

1. $t \in \mathrm{Tr}^E_{\geq 2}(a)$, $\mid t \mid = \omega$ and $t[j] \models \neg\Psi$ for all $j \geq 1$. Then, since $t[0] \models \Phi$, $t[j] \models \Phi \vee \neg\Psi$ for all $j \in \mathbb{N}$. Since Φ and Ψ contain no primed variables, one also has that $(t[j], t[j+1]) \models \Phi \vee \neg\Psi$ for all $j \in \mathbb{N}$. It now follows from (3.11) that $(t[j], t[j+1]) \models A$ for all $0 \leq j < \omega$. Hence $t \models A$.
2. $t \in \mathrm{Tr}^E_{\geq 2}(a)$, $\mid t \mid < \omega$, $t[j] \models \neg\Psi$ for all $1 \leq j \leq \mid t \mid -2$ and $t[\mid t \mid -1] \models \Psi$. Again, clearly, $t[j] \models \Phi \vee \neg\Psi$ for all $0 \leq j < \mid t \mid -1$. Consequently, $(t[j], t[j+1]) \models \Phi \vee \neg\Psi$ for all $0 \leq j < \mid t \mid -1$. Hence $t \models A$ by (3.11).

Rule 3.3 Assume

$$\{\Phi\}\, P\, \{\mathrm{cond}(G)\} \models_E A$$

and

$$\{\mathrm{cond}(G)\}\, G\, \{\Psi\} \models_E A\ ,$$

and consider $t \in [\![P\,; G]\!]^E_\Psi$ such that $t[0] \models \Phi$. By definition of $\oplus$ there exist $t_1 \in [\![P]\!]^E_{\mathrm{cond}(G)}$, $t_2 \in [\![G]\!]^E_\Psi$ such that $t = t_1 \circ t_2$ and by definition of $\circ$ one distinguishes the following two sub-cases:

1. $\mid t_1 \mid = \omega$. Then $t = t_1$ and $t \models A$ by assumption $\{\Phi\}\, P\, \{\mathrm{cond}(G)\} \models_E A$.
2. $\mid t_1 \mid = n < \omega$ and, necessarily, $t_1[n-1] = t_2[0]$. By Lemma 3.1, $n \geq 2$ and $\mid t_2 \mid \geq 2$. Now $t_1[0] \models \Phi$ since $t[0] \models \Phi$ and $t_1[0] = t[0]$, whence $t_1 \models A$ by assumption $\{\Phi\}\, P\, \{\mathrm{cond}(G)\} \models_E A$. By Lemma 3.1 and $t_2[0] = t_1[n-1]$ it follows that $t_2[0] \models \mathrm{cond}(G)$. Hence $t_2 \models A$ by assumption $\{\mathrm{cond}(G)\}\, G\, \{\Psi\} \models_E A$. Thus $t \models A$ as required.

Rule 3.4 Obviously sound.

Rule 3.5 Assume

$$\{\Phi \vee b\}\, P\, \{\Psi \vee b\} \models_E A \ . \tag{3.12}$$

It will be shown that $\{\Phi \vee b\}$ loop $P\colon b\, \{\Psi\} \models_E A$, from which it will follow that $\{\Phi\}$ loop $P\colon b\, \{\Psi\} \models_E A$.

Recalling equation (3.10) it suffices to prove for all $n \geq 0$ that $t \models A$ for all $t \in T_n$ such that $t[0] \models \Phi \vee b$. The proof of this proceeds by induction on n: For T_0 the result holds vacuously since $T_0 = \emptyset$. Consider now $t \in T_{n+1}$ such that $t[0] \models \Phi \vee b$. By definition, $T_{n+1} = F(T_n) = [\![P]\!]^E_\Psi \cup ([\![P]\!]^E_b \oplus T_n)$ and thus there are two cases (suppressing irrelevant environment information):

1. $t \in [\![P]\!]_\Psi$. Then $t \models A$ follows immediately from assumption (3.12) and the fact that $[\![P]\!]_\Psi \subseteq [\![P]\!]_{\Psi \vee b}$.
2. $t \in [\![P]\!]_b \oplus T_n$. Here the proof is similar to that for rule (3.3): $t = t_1 \circ t_2$ for some $t_1 \in [\![P]\!]_b$ and some $t_2 \in T_n$. If $\mid t_1 \mid = \omega$ then $t \models A$ follows from assumption (3.12). If $\mid t_1 \mid = m < \omega$ then $t_1 \models A$ also by assumption (3.12). By Lemma 3.1, $t_1[m-1] \models b$ and hence $t_1[m-1] \models \Phi \vee b$. By definition of $\circ$ it then follows that $t_2[0] \models \Phi \vee b$ and hence $t_2 \models A$ by the induction hypothesis for T_n. Hence $t \models A$ as required.

Rules 3.6, 3.7, 3.8 & 3.9 Obviously sound.

Chapter 4

On the Coexistence of Text and Diagrams in DSLs

In many domain specific languages diagrammatic notation is used because it conforms to notations used by domain specialists before the deployment of programmable components. The aim is to lessen the possibility of error by changing as little as possible. However the switch to programmable components often means a radical change in the details of the implementation. Such changes can mean that the domain experts' interpretation of the notation diverges significantly from the actual implementation.

We explore this problem, taking programmable controllers as a specific example. The IEC 1131-3 international standard has both a diagrammatic notation and a textual language for the description of "function blocks" which are the basic components of controller programs. We take an idealised version of the textual language and of its diagrammatic counterpart and show that the diagrams capture equivalence of textual programs under a collection of equational laws.

This result establishes that diagrams relieve the programmer of the need to consider non-significant variants of programs and the match between program texts, their corresponding diagrams and their intended interpretation.

4.1 Motivation

The use of domain-specific languages is becoming ever more widespread as programmable components replace "hard-wired" systems. One feature of such languages is their choice of programming notation. Frequently this is diagrammatic and derived from earlier design practice. The use of such notation is appealing because it eases the transition to new implementation technology. However, it can introduce new risks. Any given, non-atomic diagram can be built or decomposed in a variety of ways and the notation usually has a concrete representation as a term in some language (or as a representation in the programming system). If the interpretation of the diagram is sensitive to this information it could mean that seemingly identical programs have divergent interpretations and that some of these interpretations differ from what the developer expects. This places an obligation on the language designers to establish that all methods of building the same diagram are interpreted identically. As far as we know there are few results of this kind. Milner's result on the correspondence between flowgraphs and CCS terms [103, 102] is a notable exception.

Here we present an exploration of the correspondence between diagrams and their textual form for an idealisation of IEC 1131-3 [23, 86] function blocks. Users of the function block diagrams are encouraged to think in a "circuit" paradigm [111, 135] – but the diagrams are internally translated into a textual form and are interpreted by microprocessors. The potential for divergence of the expected behaviour of diagrammatic notations is obvious. For example, failure of the associativity of diagram composition is certainly possible if care is not taken in defining the meaning of the textual form of the program or in implementing this definition.

4.2 Outline of method

Inspired by the work of Milner [102], our approach consists in:

1. Enumerating the various classes of programs and program formation operations in a signature Σ. (Σ may be left implicit if properly understood from the context.) A term algebra $\mathcal{F} = \mathbb{T}(\Sigma, X)$ over this signature formalises the abstract syntax of (textual) programs, where the elements of X stand for (the names of) pre-defined programs, available from libraries or otherwise. For function blocks, below, we take $X = \emptyset$.

2. Defining the set of valid diagrammatic representations formally as an algebra $\mathcal{D}$ over the same signature Σ. In particular, this provides a "diagrammatic interpretation" of program formation operations. Usually, $\mathcal{D}$ will be generated by a set Γ of atomic diagrams (i.e. all diagrams in $\mathcal{D}$ are constructed from those in Γ using the operations) and one is interested in proving that all valid diagrams arise in this way. The elements of the term algebra $\mathbb{T}(\Sigma, \Gamma)$ are therefore called *diagram expressions.*

3. Observing that, in general, each composite diagram in $\mathcal{D}$ will have multiple

representations in terms of diagram expressions. Intuition about diagrams induces a (usually non-empty) set E of equational laws which hold in $\mathcal{D}$.

4. Establishing $\mathcal{D}$ as a *free* Σ algebra subject to E by proving that the laws in E are *complete* for $\mathcal{D}$. (That is, whenever any two diagram expressions denote the same diagram, then they can be shown equal using the axioms in E.) This establishes an isomorphism between $\mathcal{D}$ and the quotient $\mathbb{T}(\Sigma, \Gamma)/E$ of $\mathbb{T}(\Sigma, \Gamma)$ induced by the equational laws E. Intuitively, each element of that quotient algebra collects all diagram expressions corresponding to the same diagram in $\mathcal{D}$.

5. Establishing the link between diagram expressions and the abstract syntax of programs by providing a bijection γ between Γ and $\mathcal{F}$. Intuitively, this means that every program can be uniquely represented as an atomic diagram hiding the program's internal structure and, also, that every atomic diagram is a representation of some program. Then, the properties of term algebras and the freeness of $\mathcal{D}$ assert that:

$$\mathbb{T}(\Sigma, \mathcal{F})/E \cong \mathbb{T}(\Sigma, \Gamma)/E \cong \mathcal{D} \ ,$$

   where $\cong$ is read "is isomorphic to". That is, diagrams capture precisely the equivalence classes of programs under E.

6. Interpreting the terms in $\mathcal{F}$ by means of an operational or denotational semantics and proving the soundness of the laws in E under the chosen interpretation. This establishes that the evaluation of all programs corresponding to the same diagram produce identical computations.

4.2.1 Notation

Before proceeding to the application of our method, let us introduce some frequently used notation. Let h be a function. Then $\operatorname{dom} h$ and $\operatorname{ran} h$ are the *domain* and *range* of h, respectively. We will also write $\operatorname{dom} R$ and $\operatorname{ran} R$ to mean those subsets of A and B, respectively, over which a relation $R \subseteq A \times B$ is defined.

If $X \subseteq \operatorname{dom} h$, we write $h \restriction X$ for the *restriction* of h on X and $h(X)$ for the image of X under h. Assuming that g is a function with $\operatorname{ran} g \subseteq \operatorname{dom} h$, the *composition* $h \circ g$ of g with h is then $(h \circ g)(x) \overset{\text{def}}{=} h(g(x))$. Finally, for X an arbitrary finite set, we let $|X|$ denote the number of elements in X.

Combinator	Name	Precedence
$f_1 \gg f_2$	Composition	2
$f \setminus X$	Restriction	1
$f[g]$	Relabelling	1
$f[c;\, y \to x]$	Loop	1

Table 4.1: Function block combinators

4.3 An abstract syntax for function blocks

In [140, 3] we introduce a rationalised version of the function block language, provide it with an operational semantics and argue that the rationalisation is adequate to describe most PLC programs as well as to support formal reasoning about such programs. Here we are mostly concerned with the relationship between diagrams and the textual form of the language. To compress the presentation we shall omit a detailed account of our language, although a summary of the semantics will be provided in Section 4.8. For the time being, we outline an abstract syntax corresponding to the textual representation of function blocks in IEC 1131-3.

The set of function block terms, ranged over by f, is produced by the following abstract grammar:

$$f \quad ::= \quad \mathsf{y} = e \mid f \setminus X \mid f[g] \mid f[c;\, y \to x] \mid f_1 \gg f_2$$

Composite function blocks are constructed using the combinators in Table 4.1. Every function block f has a finite number of labelled ports, each classified as either an input or an output port, and the combinators use these in their definitions. Composition assigns each output of f_2 to an input of f_1 having the same label, restriction hides (internalises) some ports, and relabelling recasts the labels of ports according to the mapping g. Looping connects an output port to an input port while specifying the initial value on the loop. In terms of IEC 1131-3 syntax, each $f[c;\, y \to x]$ corresponds to an initialised local variable over f. Schematically:

```
FUNCTION_BLOCK ‘‘F[c; Y -> X]’’
  ...
  VAR YOLD := c : ... END_VAR
BEGIN_BLOCK
  F( X := YOLD; ... );
  YOLD := F.Y
END_BLOCK
```

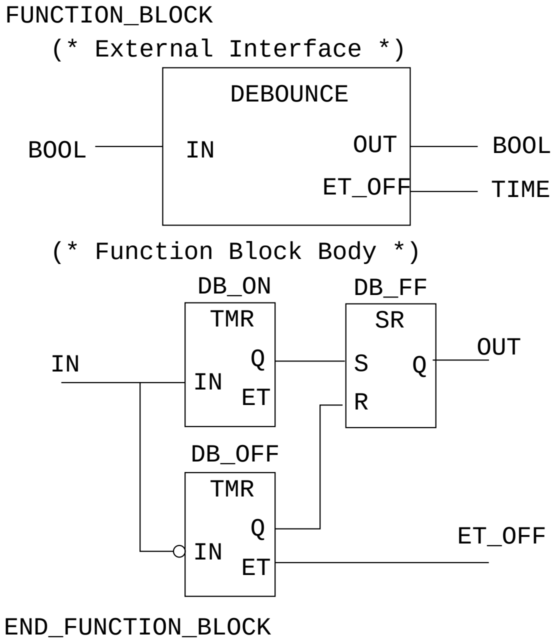


Figure 4.1: Function block DEBOUNCE. (Repeated for convenience from Figure 2.4.)

In the remaining clause, $\mathsf{y} = e$ defines the output port y to be set to the value of the expression e (which may use input ports).

Example 4.1. We illustrate the syntax by providing a translation term f_{DB} corresponding to the DEBOUNCE block of Figure 4.1. This will have a single input port labelled in and two output ports labelled out and $\mathsf{et\_off}$. The overall decomposition of f_{DB} is

$$f_{DB} \overset{\text{def}}{=} (f_{INV} \gg f_{DB\_ON} \gg f_{DB\_OFF} \gg f_{DB\_FF}) \setminus S \ ,$$

that is a restricted composition where $S = \{\mathsf{r}\}$. The f_{INV} component is a simple boolean inverter:

$$f_{INV} \overset{\text{def}}{=} \mathsf{n} = \neg \mathsf{in} \ .$$

The f_{DB_ON},f_{DB_OFF} and f_{DB_FF} terms are the instances of TMR and SR with their ports suitably relabelled to specify the desired connections. Using pairs of the form $g(\mathsf{x})/\mathsf{x}$ as a compact notation for relabellings, one has:

$$\begin{aligned} f_{DB\_ON} &\overset{\text{def}}{=} f_{TMR}[\mathsf{s}/\mathsf{q}] \setminus \mathsf{et} \\ f_{DB\_OFF} &\overset{\text{def}}{=} f_{TMR}[\mathsf{n}/\mathsf{in}, \mathsf{et\_off}/\mathsf{et}, \mathsf{r}/\mathsf{q}] \\ f_{DB\_FF} &\overset{\text{def}}{=} f_{SR}[\mathsf{out}/\mathsf{q}] \ . \end{aligned}$$

For instance, the input in of f_{TMR} in f_{DB_OFF} has been renamed to n so as to be connected to the output port n of the inverter f_{INV}. Also, the et port of f_{TMR} in f_{DB_ON} has been hidden by the restriction and therefore has been removed from the outputs of f_{DB}. Finally, the restriction $\setminus S$ internalises all those output ports which do not appear among the outputs of f_{DB}.

The SR flip-flop is a simple example of a history-dependent function block, with a possible representation for it being

$$f_{SR} \overset{\text{def}}{=} (\mathsf{q} = \mathsf{s} \vee (\mathsf{r} \wedge \mathsf{q1}))[\mathsf{ff};\, q \rightarrow q1] \ .$$

The current value of output q depends not only on the current values of inputs s and r but also on the previous value of q, fed back to the input q1 via the loop $[\mathsf{ff};\, q \rightarrow q1]$. Thus, the role of the loop linking q to q1 in f_{SR} is that of a local variable in the terminology of IEC 1131-3. In this case, the initial value for that variable is ff. ■

We write in$f$ and out$f$ respectively for the labels on the input and output ports of f. Port labels are drawn from a set *PLab*, its finite subsets being ranged over by X, Y, Z, $\ldots$. Constants belong to a set *Con*. We now make precise what we mean by a “relabelling map”:

Definition 4.1. Let $\langle X, Y\rangle$ and $\langle X', Y'\rangle$ be pairs of disjoint subsets of *PLab*. A *relabelling* g from $\langle X, Y\rangle$ to $\langle X', Y'\rangle$ is a pair $\langle g_0, g_1\rangle$ of functions $g_0 : X \rightarrow X'$ and $g_1 : Y \rightarrow Y'$ such that g_0 is surjective and g_1 is bijective. We shall write $g : \langle X, Y\rangle \rightarrow \langle X', Y'\rangle$ to denote a relabelling from $\langle X, Y\rangle$ to $\langle X', Y'\rangle$. Each relabelling g may be specified as a list of pairs of the form $g_0(x)/x$ and $g_1(y)/y$, omitting those pairs in which both labels are the same. Relabellings compose in the obvious way: given $g : \langle X, Y\rangle \rightarrow \langle X', Y'\rangle$ and $g' : \langle X', Y'\rangle \rightarrow \langle X'', Y''\rangle$, their composite $g' \circ g$ is $\langle g_0' \circ g_0, g_1' \circ g_1\rangle$. ■

We now provide an explicit description of the term algebra $\mathcal{F}$. Each carrier $F(X, Y) \in \mathcal{F}$ collects all function block terms f with in$f = X$ and out$f = Y$:

Definition 4.2. The sets $F(X, Y)$ are inductively defined as follows:

1. $\mathsf{y} = e \in F(X, \{\mathsf{y}\})$, for all expressions e whose set of port labels is X and all port labels $\mathsf{y} \notin X$.

2. Whenever $f_1 \in F(X_1, Y_1)$, $f_2 \in F(X_2, Y_2)$ and $Y_1 \cap Y_2 = \emptyset$, then $f_1 \gg f_2 \in F((X_1 \cup X_2) \setminus Y_1, Y_1 \cup Y_2)$.

3. Whenever $f \in F(X, Y)$ and $g : \langle X, Y\rangle \rightarrow \langle X', Y'\rangle$, then $f[g] \in F(X', Y')$.

4. Whenever $f \in F(X,Y)$, then $f \setminus \mathsf{y} \in F(X, Y \setminus \{\mathsf{y}\})$ for all port labels y.

5. Whenever $f \in F(X,Y)$, then $f[c; \mathsf{y} \to \mathsf{x}] \in F(X \setminus \{\mathsf{x}\}, Y)$ for any $\mathsf{x} \in X$, $\mathsf{y} \in Y$ and $c \in Con$. ■

It is routine to check that $X \cap Y = \emptyset$ for all $F(X,Y)$ in $\mathcal{F}$. That is, in well-formed function block terms the labels on the inputs and outputs are disjoint. For simplicity, we shall write $f \in \mathcal{F}$ to mean "f belongs to some $F(X,Y)$ in $\mathcal{F}$" whenever knowledge of the exact sets X and Y is immaterial.

4.4 Diagrams

Informally, a function block diagram consists of a set of nodes, each labelled with a function block term and possessing a set of ports. Each port has an *internal label* and, optionally, an *external label*. All labels are drawn from the set *PLab* of port labels. Furthermore, some input ports may be assigned a constant c from the set *Con*. Finally, ports may be connected by means of *connections*.

Definition 4.3. In the context of this chapter, a diagram

$$d = (N, F, I, O, C, \lambda, \varepsilon, \iota)$$

consists of:

1. A finite set N of nodes. Each $n \in N$ possesses two disjoint sets $I(n)$ and $O(n)$ of ports such that $I(n) \cap I(m) = \emptyset$ and $O(n) \cap O(m) = \emptyset$ whenever $n \neq m$.

2. A map $F \colon N \to \mathcal{F}$ which assigns a function block term $F(n)$ to each node $n \in N$ such that $|I(n)|{=}|\mathrm{in}F(n)|$ and $|O(n)|{=}|\mathrm{out}F(n)|$ for all $n \in N$.

3. A set of input ports $I = \bigcup_{n \in N} I(n)$ and a set of output ports $O = \bigcup_{n \in N} O(n)$.

4. A relation $C \subseteq O \times I$, the set of *connections*. Each connection $(o, i) \in C$ represents a directed edge from port o to port i.

5. A total map $\lambda : I \cup O \to PLab$, the *internal label map*, such that $\lambda(I(n)) = \mathrm{in}F(n)$ and $\lambda(O(n)) = \mathrm{out}F(n)$ for all $n \in N$.

6. A partial map ε from $I \cup O$ to *PLab*, the *external label map*.

7. A partial map ι from I to *Con*, the *port initialisation map*. ■

The set in$d$ of *input labels* in a diagram d is defined to be $\{\varepsilon(p) \mid p \in I \cap \operatorname{dom} \varepsilon\}$, that is the set of all external labels assigned to input ports. Similarly, the set of *output labels* out$d$ of d is the set of all external labels assigned to output ports: $\{\varepsilon(p) \mid p \in O \cap \operatorname{dom} \varepsilon\}$. Ports with external labels will be referred to as the *external* ports, whereas those possessing internal labels only will be called the *internal* ports.

Remark. *There is a natural notion of isomorphism for diagrams: d and d' are isomorphic if there exist bijections $\nu : N \to N'$ and $\pi : I \cup O \to I' \cup O'$ such that $\pi(p) \in I'(\nu(n))$ iff $p \in I(n)$, $\pi(p) \in O'(\nu(n))$ iff $p \in O(n)$, $F'(\nu(n)) = F(n)$, $(\pi(p), \pi(p')) \in C'$ iff $(p, p') \in C$, $\lambda'(\pi(p)) = \lambda(p)$, $\varepsilon'(\pi(p)) = \varepsilon(p)$ and $\iota'(\pi(p)) = \iota(p)$ for all nodes n and ports p, p' of d. All pertinent information about diagrams resides in the labellings of their nodes and ports, not in the nature of the (mathematical) objects one chooses as the nodes and ports themselves. We will therefore treat isomorphic diagrams as identical.*

4.4.1 From terms to diagrams

Every term $f \in \mathcal{F}$ with in$f = \{\mathsf{x}_1, \ldots, \mathsf{x}_m\}$ and out$f = \{\mathsf{y}_1, \ldots, \mathsf{y}_{m'}\}$ may now be represented by a single-node diagram $\gamma(f)$ as shown in Figure 4.2. Formally, γ is an injective function from $\mathcal{F}$ into diagrams: Let $\{p_1, \ldots, p_m\}$, $\{p'_1, \ldots, p'_{m'}\}$ be two disjoint sets of ports. Then, $\gamma(f) \overset{\text{def}}{=} (N, F, I, O, C, \lambda, \varepsilon, \iota)$, where

$$\begin{aligned}
N &= \{n\} \quad \text{is a singleton} \\
F(n) &= f \\
I &= \{p_i \mid i = 1, \ldots, m\} \\
O &= \{p'_i \mid i = 1, \ldots, m'\} \\
C &= \emptyset \\
\lambda &= \{(p_i, \mathsf{x}_i) \mid i = 1, \ldots, m\} \cup \{(p'_i, \mathsf{y}_i) \mid i = 1, \ldots, m'\} \\
\varepsilon &= \lambda \\
\iota &= \emptyset\ .
\end{aligned}$$

As we shall see shortly, every function block diagram can be generated from those in ran γ. To remind ourselves of this, we shall write Γ in place of the less mnemonic ran γ.

4.4.2 Proper diagrams

Definition 4.3 is, for reasons of simplicity, quite general. Not every diagram permitted by this definition is a proper function block diagram (although each

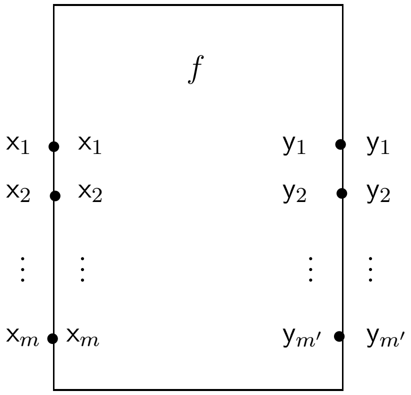


Figure 4.2: A single-node diagram of function block f.

$\gamma(f)$ certainly is).

Informally, a diagram d is proper if the following additional constraints are met:

1. Each input port can be connected to at most one output port.

2. All output ports have *distinct external labels*.

3. No input port is given the same external label as any output port, that is $\text{in}d \cap \text{out}d = \emptyset$.

4. Only unconnected input ports may have external labels.

5. Only connected input ports may be initialised.

Our aim is to define an algebra of proper function block diagrams by choosing a suitable set of operations corresponding to the combinators of Table 4.1. The carrier set of this algebra will then be precisely the set of diagrams generated from Γ using these operations. We begin by first identifying this set and postpone the introduction of the operations until the next section.

Definition 4.4. A diagram $d = (N, F, I, O, C, \lambda, \varepsilon, \iota)$ is called *proper*, iff

1. C^{-1} is a *function*

2. $\varepsilon \upharpoonright O$ is an injection

3. $\varepsilon(O) \cap \varepsilon(I) = \emptyset$

4. $\text{dom}\,(\varepsilon \upharpoonright I) = I \setminus \text{ran}\, C$; and

5. $\text{dom}\,\iota \subseteq \text{ran}\, C$. ■

Easy consequences of the above definition are that, in all proper diagrams d, $\varepsilon(\operatorname{ran} C) = \emptyset$ and also $\operatorname{dom} \iota \cap \operatorname{dom} \varepsilon = \emptyset$.

4.5 An algebra of diagrams

This section completes the definition of our algebra of diagrams by introducing a set of operations corresponding to those in the programming language. Each expression in the algebra describes a particular way of building a diagram and thus the operations are interpreted in terms of the diagrams they construct. The choice of the carrier set made in the previous section is finally justified by demonstrating that all diagrams so constructed are proper.

Definition 4.5 (Diagram Composition). Let $d_i = (N_i, F_i, I_i, O_i, C_i, \lambda_i, \varepsilon_i, \iota_i)$, $i = 1, 2$, be diagrams with $N_1 \cap N_2 = \emptyset$ and $\mathrm{out}d_1 \cap \mathrm{out}d_2 = \emptyset$. Then,

$$d_1 \gg d_2 \overset{\text{def}}{=} (N, F, I, O, C, \lambda, \varepsilon, \iota)$$

is the diagram given by:

$$\begin{array}{lll} N = N_1 \cup N_2 \ , & F = F_1 \cup F_2 \ , & I = I_1 \cup I_2 \ , \\ O = O_1 \cup O_2 \ , & C = C_1 \cup C_2 \cup \widehat{C} \ , & \lambda = \lambda_1 \cup \lambda_2 \ , \end{array}$$
$$\iota = \iota_1 \cup \iota_2 \ , \quad \varepsilon = \varepsilon_1 \cup (\varepsilon_2 \setminus \{(p, \mathsf{x}) \mid p \in \operatorname{ran} \widehat{C}\})$$

where $\widehat{C} = \{(p, p') \in O_1 \times I_2 \mid \varepsilon_1(p) = \varepsilon_2(p') \in \mathit{PLab}\}$. ■

Definition 4.6 (Diagram Restriction). For all $d = (N, F, I, O, C, \lambda, \varepsilon, \iota)$ and $\mathsf{x} \in \mathit{PLab}$,

$$d \setminus \mathsf{x} \overset{\text{def}}{=} (N, F, I, O, C, \lambda, \varepsilon', \iota) \ ,$$

where $\varepsilon' = \varepsilon \setminus \{(p, \mathsf{x}) \mid p \in O\}$. ■

Definition 4.7 (Diagram Relabelling). For all $d = (N, F, I, O, C, \lambda, \varepsilon, \iota)$ and relabellings g of $\langle \mathrm{in}d, \mathrm{out}d \rangle$,

$$d[g] \overset{\text{def}}{=} (N, F, I, O, C, \lambda, \varepsilon', \iota) \ ,$$

where $\varepsilon' = (g_0 \circ (\varepsilon \restriction I)) \cup (g_1 \circ (\varepsilon \restriction O))$. ■

Definition 4.8 (Diagram Loop). For all $d = (N, F, I, O, A, \lambda, \varepsilon, \iota)$, $c \in \mathit{Con}$ and $\mathsf{x} \in \mathrm{in}d$, $\mathsf{y} \in \mathrm{out}d$. Then

$$d[c;\, y \to x] \overset{\text{def}}{=} (N, F, I, O, C \cup \widehat{C}, \lambda, \varepsilon', \iota') \ ,$$

where $\widehat{C} = \{(p, p') \mid p \in O, p' \in I, \varepsilon(p) = \mathsf{y},\ \varepsilon(p') = \mathsf{x}\}$, $\varepsilon' = \varepsilon \setminus \{(p, \mathsf{x}) \mid p \in I\}$ and $\iota' = \iota \cup \{(p, c) \mid p \in \operatorname{ran} \widehat{C}\}$. ■

In words, the composition $d_1 \gg d_2$ connects those output ports of d_1 to those input ports of d_2 having matching external labels. The external labels of the inputs thus connected are subsequently removed. The restriction in $d \setminus \mathsf{y}$ removes all occurrences of the external label y from the output ports of d. Relabelling applies the relabelling function g to recast the external labels of d. Finally, $d[c;\, y \rightarrow x]$ connects the output port of d having external label y to all input ports of d having external label x. The input ports so connected have their external labels subsequently removed and are initialised with the constant c.

Let now $\mathcal{D}$ be the algebra generated by Γ, that is the least set containing Γ that is closed under the above operations. Expressions over this algebra, i.e. well-formed expressions built from the elements of Γ using the operations, will be called *diagram expressions*. The name stems from the fact that diagram expressions provide a syntactic means to describe *every* diagram in $\mathcal{D}$, with most non-atomic diagrams (i.e. diagrams in $\mathcal{D} \setminus \Gamma$) having multiple descriptions. As a notational convention, we will use f to stand for $\gamma(f)$ in diagram expressions and use w to range over diagram expressions.

Lemma 4.1. *The diagram operations of Composition, Restriction, Relabelling and Loop preserve properness.*

Proof. Routine from the definitions. ■

With the help of this lemma, one may now prove that the proper diagrams are precisely the diagrams in $\mathcal{D}$.

Proposition 4.1. *A diagram is in $\mathcal{D}$ iff it is proper.*

Proof. To establish the forward implication one needs to show that all diagrams in Γ are proper, which holds by definition of γ. Then by Lemma 4.1, all diagrams in $\mathcal{D}$ are proper.

For the converse implication assume $d = (N, F, I, O, C, \lambda, \varepsilon, \iota)$ proper.

1. Add distinct *fresh* labels[1] $\mathsf{z}_1, \ldots, \mathsf{z}_k$, one for each output port $p_1, \ldots, p_k \in O \setminus \operatorname{dom} \varepsilon$ to form diagram $d' = (N, F, I, O, C, \lambda, \varepsilon', \iota)$ with

   $$\varepsilon' = \varepsilon \cup \{(p_i, \mathsf{z}_i) \mid i \in \{1, \ldots, k\}\} \ .$$

   Clearly, d' is proper and

   $$d = d' \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k \ .$$

[1] i.e. not occurring among the input or output labels of d

2. Now, for each initialised input port $p_j \in \operatorname{dom} \iota$, let $p'_j = C^{-1}(p_j)$ be the *unique* output port to which p_j is connected via a connection in C. Such an unique output port always exists since C^{-1} is a function and $\operatorname{dom} \iota \subseteq \operatorname{dom} C^{-1}$. Furthermore, let y_j be the *unique* external label assigned to p'_j under ε'. (The existence of y_j is guaranteed by the totality of ε' and its uniqueness by the requirement that $\varepsilon \restriction O$ — and hence $\varepsilon' \restriction O$ — is an injection.) Add now fresh labels x_j, one for each p_j, and remove all connections in C to the p_j's to form $d'' = (N, F, I, O, C'', \lambda, \varepsilon'', \emptyset)$ with

   $$C'' = C \setminus \{(p'_j, p_j) \mid p_j \in \operatorname{dom} \iota\} \ ,$$

   and

   $$\varepsilon'' = \varepsilon' \cup \{(p_j, \mathsf{x}_j) \mid p_j \in \operatorname{dom} \iota\} \ .$$

   Clearly then

   $$d' = d''[c_1; \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c_n; \mathsf{y}_n \to \mathsf{x}_n] \ ,$$

   where $n = \mid \operatorname{dom} \iota \mid$ and $c_j = \iota(p_j)$, $j = 1, \ldots, n$. Furthermore, it is not hard to verify that d'' is also proper.

3. Now consider each node n_i in d'' and let $f_i = F(n_i)$. The map ε'' is defined on every port of d'' except those input ports that are connected, that is those ports $p \in \operatorname{ran} C''$. However, ε'' extends to a *total* map $\eta : I \cup O \to \mathit{PLab}$ as follows:

   $$\eta = \varepsilon'' \cup \{(p, \varepsilon''(p')) \mid (p', p) \in C''\} \ .$$

   Since $\varepsilon'' \restriction O$ is an injection and $\varepsilon''(I) \cap \varepsilon''(O) = \emptyset$, it follows that $\eta \restriction O$ is an injection and $\eta(I(n_i)) \cap \eta(O(n_i)) = \emptyset$. Now, for each n_i define a relabelling g_i by

   $$g_i = \{\eta(p)/\lambda(p) \mid p \in P(n_i)\} \ ,$$

   that maps the internal labels of ports in n_i to their corresponding labels under η. By the properties of η and the definition of λ, it follows that each $g_i \restriction \mathrm{out} f_i$ is an injection and that $g_i(\mathrm{in} f_i) \cap g_i(\mathrm{out} f_i) = \emptyset$. Moreover, it is not hard to see that

   $$d'' = f_1[g_1] \gg (f_2[g_2] \gg (\ldots))$$

   whence we obtain the diagram expression for d:

   $$d = (f_1[g_1] \gg (f_2[g_2] \gg (\ldots)))[c_1; \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c_n; \mathsf{y}_n \to \mathsf{x}_n] \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k.$$

■

Thus, *every* proper diagram can be described by at least one diagram expression involving operations which correspond to those in the textual representation.

4.6 Equational Laws

In this section we develop an equational system E for our algebra of diagrams. The system comprises seventeen equational laws which provide the means to link diagrams to the textual form of the language. To establish the result that there is no deviation between diagrammatic and textual programs, one must also establish the validity of these laws in the semantics of the programming language. This is done in Section 4.8.

Proposition 4.2 (Equational Laws). *The identities in Table 4.2 hold in $\mathcal{D}$, whenever $d, d_i, \ldots$ are diagrams such that both sides of each equation are well-defined.*

Proof. Routine from the definitions. ■

The following is a useful corollary of Proposition 4.2.

Corollary 4.1. *The following identities hold whenever both sides are defined:*

1. $d \setminus \mathsf{y} = d[\mathsf{z}/\mathsf{y}] \setminus \mathsf{z}$
2. $d \setminus \mathsf{x_1} \ldots \setminus \mathsf{x_n} = d[\mathsf{z_1}/\mathsf{x_1}, \ldots, \mathsf{z_n}/\mathsf{x_n}] \setminus \mathsf{z_1} \ldots \setminus \mathsf{z_n}$ *if the* z_i *are distinct.*
3. $d[c;\, \mathsf{y} \rightarrow \mathsf{x}][g] = d[g][c;\, g(\mathsf{y}) \rightarrow g(\mathsf{x})]$ *if* g_0 *is an injection.* ■

4.7 Completeness

Reverting to the question of which diagram expressions describe the same diagram, we demonstrate that the equational system E of the previous section is *complete* for $\mathcal{D}$. In other words, any two diagram expressions describing the same diagram may be proved equal from laws (E1)–(E16). We shall write $E \vdash w = w'$ to mean "w and w' may be proved equal from E". The proof of the completeness theorem is vastly simplified if one observes that any diagram expression can be converted using the laws into an equivalent *normal form*.

Definition 4.9 (Normal Form). A well-formed diagram expression w is in *normal form* if

$$w \stackrel{\text{def}}{=} (f_1[g_1] \gg \ldots \gg f_m[g_m])[c_1;\, \mathsf{y}_1 \rightarrow \mathsf{x}_1] \ldots [c_n;\, \mathsf{y}_n \rightarrow \mathsf{x}_n] \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k \ ,$$

where

(E1) $(d_1 \gg d_2) \gg d_3 = d_1 \gg (d_2 \gg d_3)$

(E2) $d_1 \gg d_2 = d_2 \gg d_1 \quad \text{if } \mathrm{out}d_1 \cap \mathrm{in}d_2 = \emptyset \text{ and } \mathrm{out}d_2 \cap \mathrm{in}d_1 = \emptyset$

(E3) $(d_1[c;\, \mathsf{y} \to \mathsf{x}]) \gg d_2 = (d_1 \gg d_2)[c;\, \mathsf{y} \to \mathsf{x}] \quad \text{if } \mathsf{y} \notin \mathrm{out}d_2 \text{ and } \mathsf{x} \notin \mathrm{in}d_2$

(E4) $d_1 \gg (d_2[c;\, \mathsf{y} \to \mathsf{x}]) = (d_1 \gg d_2)[c;\, \mathsf{y} \to \mathsf{x}] \quad \text{if } \mathsf{y} \notin \mathrm{out}d_1 \text{ and } \mathsf{x} \notin \mathrm{in}d_1$

(E5) $d \setminus \mathsf{y} = d \quad \text{if } \mathsf{y} \notin \mathrm{out}d$

(E6) $d \setminus \mathsf{x} \setminus \mathsf{y} = d \setminus \mathsf{y} \setminus \mathsf{x}$

(E7) $d_1 \setminus \mathsf{y} \gg d_2 = (d_1 \gg d_2) \setminus \mathsf{y} \quad \text{if } \mathsf{y} \notin \mathrm{out}d_2 \cup (\mathrm{in}d_2 \cap \mathrm{out}d_1)$

(E8) $d_1 \gg d_2 \setminus \mathsf{y} = (d_1 \gg d_2) \setminus \mathsf{y} \quad \text{if } \mathsf{y} \notin \mathrm{out}d_1$

(E9) $d[\,] = d$

(E10) $d[g][g'] = d[g' \circ g]$

(E11) $(d \setminus \mathsf{y})[g] = d[g, \mathsf{z}/\mathsf{y}] \setminus \mathsf{z}$

(E12) $(d_1 \gg d_2)[g] = d_1[g_1] \gg d_2[g_2] \quad \text{if } g_i = g \restriction \langle \mathrm{in}d_i, \mathrm{out}d_i \rangle$

(E13) $d[c;\, \mathsf{y} \to \mathsf{x}][g] = d[g, \mathsf{z}/\mathsf{x}][c;\, g(\mathsf{y}) \to \mathsf{z}] \quad \text{if } \mathsf{z} \notin \mathrm{ran}\, g_0$

(E14) $d \setminus \mathsf{z}[c;\, \mathsf{y} \to \mathsf{x}] = d[c;\, \mathsf{y} \to \mathsf{x}] \setminus \mathsf{z}$

(E15) $d[c;\, \mathsf{y} \to \mathsf{x}][c';\, \mathsf{y}' \to \mathsf{x}'] = d[c';\, \mathsf{y}' \to \mathsf{x}'][c;\, \mathsf{y} \to \mathsf{x}]$

(E16) $d[c;\, \mathsf{y} \to \mathsf{z}][c;\, \mathsf{y} \to \mathsf{x}] = d[\mathsf{x}/\mathsf{z}][c;\, \mathsf{y} \to \mathsf{x}]$

Table 4.2: The equational laws for the algebra of function block diagrams.

- the z_i are *distinct* and all occur in some of the output sorts $\mathrm{out}(f_i[g_i])$, $i = 1, \ldots, m$.

- the (y_i, c_i) are all *distinct* pairs.

(By the well-formedness of w it follows that the x_i are also distinct.) ■

Lemma 4.2 (Normal Form Lemma). *For every diagram expression w, there is a normal form $w^\star$ such that $E \vdash w = w^\star$.*

Proof. We outline the phases of the transformation of w into $w^\star$, writing w_i for the resulting expression at the end of phase i.

1. Use (E5) where applicable to remove any unnecessary restrictions; that is restrictions $\backslash\mathsf{z}$ where z is not in the output sort of the subexpression qualified by the restriction.

2. Some of the labels z in the remaining restrictions $\backslash\mathsf{z}$ may not be distinct from each other, or may occur in either

$$X = \{\mathsf{x} \mid \text{there is a loop } [c;\, \mathsf{y} \rightarrow \mathsf{x}] \text{ in } w_1\}\ ,$$

   or in$w \cup$ out$w$. In this case, pick up a fresh[2] label s for each offending restriction $\backslash\mathsf{z}$ and use Corollary 4.1(1) to replace $\backslash\mathsf{z}$ by $[\mathsf{s}/\mathsf{z}] \setminus \mathsf{s}$. All labels restricted over in w_2 are now distinct from each other, and do not occur in either in$w \cup$ out$w$ or X.

3. Use (E11), (E12) and (E13) to move all relabellings in w_2 innermost. In particular, (E11) can always be applied as a result of the work done in phase 1, care must be taken in choosing z so as to preserve the work of phase 2. Finally, use (E10) to coalesce innermost relabellings.

4. Move all restrictions $\backslash\mathsf{z}$ in w_3 outermost, past any occurrences of compositions and loops. The only slightly problematic case is that of composition, which may be analysed as follows:

   - Suppose that $\backslash\mathsf{z}$ occurs in the context $d' \gg (d'' \setminus \mathsf{z})$. Now, $\mathsf{z} \notin$ outd' since otherwise $\mathsf{z} \in$ out$w$ or $\backslash\mathsf{z}$ occurs further out in w_3, and these possibilities have been eliminated during phase 2.

[2] In the context of this proof, a *fresh* label is one not occurring anywhere in the entire expression being manipulated.

- Suppose that $\backslash\mathsf{z}$ occurs in the context $(d' \setminus \mathsf{z}) \gg d''$. Again, one has that $\mathsf{z} \notin \mathrm{out}d'' \cup (\mathrm{in}d'' \cap \mathrm{out}d')$. Firstly, $\mathsf{z} \notin \mathrm{out}d''$ for reasons similar to those exhibited in the previous case. Secondly, $\mathsf{z} \notin (\mathrm{in}d'' \cap \mathrm{out}d')$ since otherwise either $\mathsf{z} \in \mathrm{in}w$, or there is a loop $[c;\, \mathsf{y} \rightarrow \mathsf{z}]$ further out which hides the input label z. Again, phase 2 has precluded these possibilities.

Hence, one may use (E7), (E8) and (E14) freely to move all restrictions $\backslash\mathsf{z}$ in w_3 outermost.

5. Move all loops $[c;\, \mathsf{y} \rightarrow \mathsf{x}]$ in w_4 outside all occurrences of composition. Suppose a loop occurs in the context $(d'[c;\, y \rightarrow x]) \gg d''$ and $\mathsf{x} \in \mathrm{in}d''$ so that (E3) cannot be applied directly[3]. Then one may use the following sequence of transformations to move the loop out:

$$\begin{array}{cl} & (d'[c;\, \mathsf{y} \rightarrow \mathsf{x}]) \gg d'' \\ = & \qquad \langle \text{using (E9)} \rangle \\ & (d'[c;\, \mathsf{y} \rightarrow \mathsf{x}][\,]) \gg d'' \\ = & \qquad \langle \text{using (E13) with } \mathsf{z} \notin \mathrm{in}f'' \rangle \\ & (d'[\mathsf{z}/\mathsf{x}][c;\, \mathsf{y} \rightarrow \mathsf{z}]) \gg f'' \\ = & \qquad \langle \text{using (E3)} \rangle \\ & (d'[\mathsf{z}/\mathsf{x}] \gg d'')[c;\, \mathsf{y} \rightarrow \mathsf{z}] \end{array}$$

In particular, z must be distinct from any restriction and any input label occurring in a loop of w_4 in order to preserve the work of the previous phases. A similar course of action is taken when the loop occurs in the context $d' \gg (d''[c;\, \mathsf{y} \rightarrow \mathsf{x}])$.

6. Coalesce relabellings in w_5 using (E10).

7. w_6 now has the required form, except that some of the (y_i, c_i) may not be distinct. In such a case, follow the procedure suggested by the following

[3] Obviously, one has that $\mathsf{y} \notin \mathrm{out}d''$ since $\mathsf{y} \in \mathrm{out}d'$ by definition and $\mathrm{out}d' \cap \mathrm{out}d'' = \emptyset$ by the properness of the diagram being manipulated.

example, where w' is a composite of diagrams of the form f_i or $f_i[g_i]$:

$$\begin{array}{rl} & w'[c_1;\, \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c;\, \mathsf{y} \to \mathsf{x}_i] \ldots [c;\, y \to \mathsf{x}_j] \ldots [c_n;\, \mathsf{y}_n \to \mathsf{x}_n] \\ = & \qquad \langle \text{repeatedly using (E15)} \rangle \\ & w'[c;\, y \to x_i][c;\, y \to x_j][c_1;\, \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c_n;\, \mathsf{y}_n \to \mathsf{x}_n] \\ = & \qquad \langle \text{using (E16)} \rangle \\ & w'[\mathsf{x}_j/\mathsf{x}_i][c;\, \mathsf{y} \to \mathsf{x}_j][c_1;\, \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c_n;\, \mathsf{y}_n \to \mathsf{x}_n] \end{array}$$

Now move the generated relabelling into w' using (E12) and coalesce innermost relabellings. Repeat the above procedure as many times as necessary. If, finally, some atomic diagram f is not qualified by a relabelling, use (E9) to replace it with $f[\,]$. The resulting expression is $w^\star$, the required normal form. ■

The following auxiliary result concerns the special case in which two diagram expressions constructing the same diagram only involve compositions and relabellings of atomic diagrams.

Lemma 4.3. *For all diagram expressions*

$$w \overset{\text{def}}{=} f_1[g_1] \gg \ldots \gg f_m[g_m]$$

and

$$w' \overset{\text{def}}{=} f'_1[g'_1] \gg \ldots \gg f'_{m'}[g'_{m'}]$$

representing the same diagram d, $E \vdash w = w'$.

Proof. Since $d = w = w'$, it follows that the atomic diagrams f_k, f'_k and the nodes n_k of d are in bijection, whence $m' = m$. Without loss of generality, assume the bijection between the n_k and f_k to be $n_k \mapsto f_k$. Since composition is not fully commutative, however, there will in general be a permutation $j_1, \ldots, j_m$ of $1, \ldots, m$ such that the bijection between the n_k and f'_k is $n_k \mapsto f'_{j_k}$.

Also, from the properness of d follows that the $f_k[g_k]$ (respectively, the $f'_k[g'_k]$) are all *distinct* from each other. For supposing the contrary, d would have two external output ports having the same label. Hence, $f_k = f'_{j_k}$. It follows that one must also have $g_k = g'_{j_k}$, otherwise w and w' would not represent the same diagram.

We can now restate our goal as follows: for all diagram expressions

$$w \overset{\text{def}}{=} f_1[g_1] \gg \ldots \gg f_m[g_m]$$

and

$$w' \overset{\text{def}}{=} f'_1[g'_1] \gg \ldots \gg f'_m[g'_m]$$

such that $d = w = w'$ and $f_i[g_i] \neq f_j[g_j]$, $f'_i[g'_i] \neq f'_j[g'_j]$ for all $i \neq j$, $E \vdash w = w'$.

The proof of this is by induction on the number m of components in w and w'. For $m = 1$ one trivially has $E \vdash w = w'$. Now, assume that the induction hypothesis holds for all expressions w, w' of length m describing the same diagram and prove the result for the case of $m + 1$, taking

$$w \overset{\text{def}}{=} f_1[g_1] \gg \ldots \gg f_{m+1}[g_{m+1}]$$

and

$$w' \overset{\text{def}}{=} f'_1[g'_1] \gg \ldots \gg f'_{m+1}[g'_{m+1}] \ .$$

Consider $f_1[g_1]$ and let k be the unique index such that $f_1[g_1] = f'_k[g'_k]$. Since $f_1[g_1]$ is the "leftmost" component of w, none of the input ports of the corresponding node n_1 in d will be connected. Consequently, as $f'_k[g'_k]$ represents the same node n_1 in w', one has that

$$\text{in}(f'_k[g'_k]) \cap \bigcup_{j=1}^{k-1} \text{out}(f'_j[g'_j]) = \emptyset \ .$$

Hence, one may use (E1) and (E2) to transform w' as follows:

$$\begin{aligned} w' &= ((f'_1[g'_1] \ldots \gg f'_{k-1}[g'_{k-1}]) \gg f'_k[g'_k]) \gg (f'_{k+1}[g'_{k+1}] \gg \ldots \gg f'_{m+1}[g'_{m+1}]) \\ &= f'_k[g'_k] \gg (f'_1[g'_1] \gg \ldots \gg f'_{k-1}[g'_{k-1}] \gg f'_{k+1}[g'_{k+1}] \gg \ldots \gg f'_{m+1}[g'_{m+1}]) \\ &= f'_k[g'_k] \gg w'_1 \end{aligned}$$

where

$$w'_1 \overset{\text{def}}{=} f'_1[g'_1] \gg \ldots \gg f'_{k-1}[g'_{k-1}] \gg f'_{k+1}[g'_{k+1}] \gg \ldots \gg f'_{m+1}[g'_{m+1}] \ .$$

Now, if one lets

$$w_1 \overset{\text{def}}{=} f_2[g_2] \gg \ldots \gg f_{m+1}[g_{m+1}] \ ,$$

then it is not hard to see that, because $f_1[g_1] = f'_k[g'_k]$, both w_1 and w'_1 represent the *same* diagram d_1 and that they are both of length m. By applying the induction hypothesis, one now gets that $E \vdash w_1 = w'_1$. Hence, $E \vdash w = w'$ as required. ■

Lemma 4.4. *Consider the expressions*

$$w \overset{\text{def}}{=} w_1[c_1;\, \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c_n;\, \mathsf{y}_n \to \mathsf{x}_n]$$

and

$$w' \overset{\text{def}}{=} w'_1[c'_1;\, \mathsf{y'}_1 \to \mathsf{x'}_1] \ldots [c'_{n'};\, \mathsf{y'}_{n'} \to \mathsf{x'}_{n'}]$$

where

- *w_1 and w'_1 contain no instances of loops*
- *the pairs (y_i, c_i) are distinct; and*
- *the pairs $(\mathsf{y'}_j, c'_j)$ are distinct.*

If both w and w' denote the same diagram d then the (y_i, c_i) are in bijection with the $(\mathsf{y'}_j, c'_j)$, whence $n' = n$. That is, there exists a permutation b of $1, \ldots, n$ such that $\mathsf{y'}_{b(k)} = \mathsf{y}_k$ and $c'_{b(k)} = c_k$. Moreover,

$$d_1 = d'_1[\mathsf{x}_1/\mathsf{x'}_{b(1)}, \ldots, \mathsf{x}_n/\mathsf{x'}_{b(n)}]$$

where d_1 and d'_1 are the diagrams denoted by w_1 and w'_1 respectively.

Proof. Consider the set of loop connections in d:

$$C_L = \{(p, p') \in C_2 \mid p \in \operatorname{dom} \varepsilon_2, p' \in \operatorname{dom} \iota_2\} \enspace .$$

C_L can be further partitioned into sets

$$C(\mathsf{l}, k) = \{(p, p') \in C_L \mid \varepsilon(p) = \mathsf{l},\ \iota(p') = k\} \enspace .$$

In words, each connection in $C(\mathsf{l}, k)$ emanates from the same output port labelled l and terminates at some internalised input port initialised with constant k. These sets are clearly disjoint as there can be no output port with more than one external label nor a multiply initialised input port.

It follows from the disjointness assumptions and

$$d = d_1[c_1;\, \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c_n;\, \mathsf{y}_n \to \mathsf{x}_n] = d'_1[c'_1;\, \mathsf{y'}_1 \to \mathsf{x'}_1] \ldots [c'_n;\, \mathsf{y'}_{n'} \to \mathsf{x'}_{n'}]$$

that the required bijection exists such that $(\mathsf{y}_i, c_i) = (\mathsf{y'}_{b(i)}, c'_{b(i)})$. ■

Proposition 4.3 (Completeness). *Whenever w, w' denote the same diagram, $E \vdash w = w'$.*

Proof. Given Lemma 4.2 it now suffices to show that, if $d = w = w'$ and w, w' are in normal form, then $E \vdash w = w'$. Suppose then that,

$$w \overset{\text{def}}{=} \underbrace{(\overbrace{f_1[g_1] \gg \ldots \gg f_m[g_m]}^{w_1})[c_1;\, \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c_n;\, \mathsf{y}_n \to \mathsf{x}_n]}_{w_2} \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k \ ,$$

$$w' \overset{\text{def}}{=} \underbrace{(\overbrace{f'_1[g'_1] \gg \ldots \gg f'_{m'}[g'_{m'}]}^{w'_1})[c'_1;\, \mathsf{y}'_1 \to \mathsf{x}'_1] \ldots [c'_{n'};\, \mathsf{y}'_{n'} \to \mathsf{x}'_{n'}]}_{w'_2} \setminus \mathsf{z}'_1 \ldots \setminus \mathsf{z}'_{k'} \ ,$$

and let d_2, d'_2 be the diagrams denoted by w_2, w'_2, respectively. Since

$$d = d_2 \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k = d'_2 \setminus \mathsf{z}'_1 \ldots \setminus \mathsf{z}'_{k'} \ ,$$

the sets $\{\mathsf{z}_1, \ldots, \mathsf{z}_k\}$ and $\{\mathsf{z}'_1, \ldots, \mathsf{z}'_{k'}\}$ are in bijection with the internalised output ports of d. (Each restriction internalises *exactly* one output port of d.) It follows that $k = k'$ and

$$d_2 = d'_2[\mathsf{z}_1 \ldots \mathsf{z}_k / \mathsf{z}'_1 \ldots \mathsf{z}'_k] \ . \tag{4.1}$$

Using Corollary 4.1(2), one obtains:

$$w'_2[\mathsf{z}_1 \ldots \mathsf{z}_k / \mathsf{z}'_1 \ldots \mathsf{z}'_k] \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k = w'$$

Using Corollary 4.1(3), the fact that the x'_i are distinct from the z'_i (by well-formedness), and (E12), one further obtains:

$$\begin{aligned}
& w'_2[\mathsf{z}_1 \ldots \mathsf{z}_k / \mathsf{z}'_1 \ldots \mathsf{z}'_k] \\
\overset{\text{def}}{=}\ & w'_1[c'_1;\, \mathsf{y}'_1 \to \mathsf{x}'_1] \ldots [c'_{n'};\, \mathsf{y}'_{n'} \to \mathsf{x}'_{n'}][\mathsf{z}_1 \ldots \mathsf{z}_k / \mathsf{z}'_1 \ldots \mathsf{z}'_k] \\
=\ & w'_1[\mathsf{z}_1 \ldots \mathsf{z}_k / \mathsf{z}'_1 \ldots \mathsf{z}'_k][c'_1;\, \mathsf{y}''_1 \to \mathsf{x}'_1] \ldots [c'_{n'};\, \mathsf{y}''_{n'} \to \mathsf{x}'_{n'}] \\
=\ & w''_1[c'_1;\, \mathsf{y}''_1 \to \mathsf{x}'_1] \ldots [c'_{n'};\, \mathsf{y}''_{n'} \to \mathsf{x}'_{n'}]
\end{aligned}$$

where $w''_1 \overset{\text{def}}{=} f'_1[g''_1] \gg \ldots \gg f'_{m'}[g''_{m'}]$, and each g''_i is the composition of g'_i with the restriction of $(\mathsf{z}_1 \ldots \mathsf{z}_k / \mathsf{z}'_1 \ldots \mathsf{z}'_k)$ to f_i. Moreover, the pairs (y''_j, c'_j) are distinct. (The (y'_j, c'_j) are distinct, since w' is in normal form, and the relabelling $\mathsf{z}'_i \mapsto \mathsf{z}_i$ is one-to-one.)

By (4.1), both $w''_1[c'_1;\, \mathsf{y}''_1 \to \mathsf{x}'_1] \ldots [c'_{n'};\, \mathsf{y}''_{n'} \to \mathsf{x}'_{n'}]$ and w_2 above denote d_2, so Lemma 4.4 applies and establishes a bijection between the (y_j, c_j) and the (y''_j, c'_j) (whence $n' = n$). Without loss of generality we may assume (via (E15) appropriately applied prior to obtaining w and w') this bijection to be $(\mathsf{y}_i, c_i) = (\mathsf{y}''_i, c'_i)$ and, consequently, that

$$d_1 = d''_1[\mathsf{x}_1 \ldots \mathsf{x}_n / \mathsf{x}'_1 \ldots \mathsf{x}'_n] \ , \tag{4.2}$$

where d''_1 is the diagram denoted by w''_1.

Now (E12) derives,

$$w''_1[\mathsf{x}_1 \ldots \mathsf{x}_n/\mathsf{x'}_1 \ldots \mathsf{x'}_n] = f'_1[g'''_1] \gg \ldots \gg f'_{m'}[g'''_{m'}] \ ,$$

where each g'''_i is the composition of g''_i with the restriction of $(\mathsf{x}_1 \ldots \mathsf{x}_n/\mathsf{x'}_1 \ldots \mathsf{x'}_n)$ to f'_i. Letting

$$w'''_1 \overset{\text{def}}{=} f'_1[g'''_1] \gg \ldots \gg f'_{m'}[g'''_{m'}] \ ,$$

(4.2) asserts that w_1 and w'''_1 both denote d_1, so Lemma 4.3 applies to establish $E \vdash w_1 = w'''_1$.

Putting it all together (in reverse order):

$$\begin{aligned}
w &= w_1[c_1;\, \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c_n;\, \mathsf{y}_n \to \mathsf{x}_n] \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k \\
&= w'''_1[c_1;\, \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c_n;\, \mathsf{y}_n \to \mathsf{x}_n] \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k \\
&= w''_1[\mathsf{x}_1 \ldots \mathsf{x}_n/\mathsf{x'}_1 \ldots \mathsf{x'}_n][c_1;\, \mathsf{y}_1 \to \mathsf{x}_1] \ldots [c_n;\, \mathsf{y}_n \to \mathsf{x}_n] \\
&\quad \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k \\
&= \ldots \qquad \langle \text{using (E13) and (E9)} \rangle \\
&= w''_1[c_1;\, \mathsf{y}_1 \to \mathsf{x'}_1] \ldots [c_n;\, \mathsf{y}_n \to \mathsf{x'}_n] \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k \\
&= w'_2[\mathsf{z}_1 \ldots \mathsf{z}_k/\mathsf{z'}_1 \ldots \mathsf{z'}_k] \setminus \mathsf{z}_1 \ldots \setminus \mathsf{z}_k \\
&= w' \ .
\end{aligned}$$

This proves $E \vdash w = w'$. ■

4.8 Connecting diagrams and programs

To complete the connection we must establish that the equations of Proposition 4.2 hold for the operational semantics of the programming language.

4.8.1 Semantics of Function Blocks

The detailed definition of the semantics is given elsewhere [140]. In general, a function block f is evaluated against a set i of input values to produce a set o of output values. As a result of evaluation, f *evolves* into a new term f' of the same type which records the new state of f. Formally, we define relations $\xrightarrow{i,o} \subseteq F(X,Y) \times F(X,Y)$, written in the form

$$f \xrightarrow{i,o} f' \ ,$$

$$\frac{i \vdash e \to c}{\mathsf{y} = e \xrightarrow{i\,,(c/\mathsf{y})} \mathsf{y} = e} \tag{4.3}$$

$$\frac{f \xrightarrow{i\,,o} f'}{f \setminus \mathsf{y} \xrightarrow{i\,,o\setminus\{\mathsf{y}\}} f' \setminus \mathsf{y}} \tag{4.4}$$

$$\frac{f \xrightarrow{i^g\,,o} f'}{f[g] \xrightarrow{i\,,o^g} f'[g]} \quad \langle i^g, o^g \rangle = \langle i, o \rangle^g \tag{4.5}$$

$$\frac{f \xrightarrow{i+(c/\mathsf{x})\,,o} f'}{f[c;\, y \to x] \xrightarrow{i\,,o} f'[o(\mathsf{y});\, y \to x]} \tag{4.6}$$

$$\frac{f_1 \xrightarrow{i_1\,,o_1} f_1' \quad f_2 \xrightarrow{i_2\,,o_2} f_2'}{f_1 \gg f_2 \xrightarrow{i\,,o_1+o_2} f_1' \gg f_2'} \quad \begin{array}{l} i_1 = i \restriction \mathrm{in} f_1 \\ i_2 = (i + o_1) \restriction \mathrm{in} f_2 \end{array} \tag{4.7}$$

Figure 4.3: Semantic rules: function block evaluation.

where $i : \mathrm{in} f \to \mathit{Con}$ and $o : \mathrm{out} f \to \mathit{Con}$. (We blur the distinction between constants and the values they denote.) Function block evaluation depends upon the evaluation of basic expressions. Writing $i \vdash e \to c$ for the evaluation of expression e to constant c under input map i, the rules for function block evaluation are as in Figure 4.3.

Specific maps i, o are given explicitly in the form

$$(c_1/\mathsf{x}_1, \ldots, c_k/\mathsf{x}_k) \;\;,$$

and we write $o \setminus \mathsf{y}$ for the map resulting by excluding y from the domain of o. The map $o + o'$ stands for the modification of o by o', having as domain the union of their domains and values given by:

$$(o + o')(\mathsf{x}) \overset{\text{def}}{=} o'(\mathsf{x}) \text{ if } \mathsf{x} \in \mathrm{dom}\, o' \text{ or } o(\mathsf{x}) \text{ otherwise.}$$

Given relabelling $g : \langle X, Y \rangle \to \langle X', Y' \rangle$ and maps $i : X' \to \mathit{Con}$, $o : Y \to \mathit{Con}$ define $\langle i, o \rangle^g$ to be the unique pair $\langle i^g, o^g \rangle$ of maps $i^g : X \to \mathit{Con}$ and $o^g : Y' \to \mathit{Con}$ such that $i^g = i \circ g_0$ and $o^g \circ g_1 = o$.

The evaluation relation $\longrightarrow$ can easily be proved *monogenic*, that is $\longrightarrow$ is a function.

Based on the semantics, one can now define a notion of *behavioural equivalence* between function blocks:

Definition 4.10. Two function blocks f_1 and f_2 of the same type are behaviourally equivalent, written $f_1 \approx f_2$, iff for all appropriate maps i and o,

1. whenever $f_1 \xrightarrow{i\,,o} f_1'$, then $f_2 \xrightarrow{i\,,o} f_2'$ and $f_1' \approx f_2'$; and
2. whenever $f_2 \xrightarrow{i\,,o} f_2'$, then $f_1 \xrightarrow{i\,,o} f_1'$ and $f_1' \approx f_2'$. ■

$\approx$ is both an equivalence and a congruence relation. The definition of $\approx$, together with the fact that evaluation preserves the syntactic structure of function block terms, gives rise to the following proof technique [140]:

Lemma 4.5. *Let f be some structural combination $\mathcal{C}(f_1, \ldots, f_n)$ of $f_1, \ldots, f_n$ and let f' be some combination $\mathcal{C}'(f_1', \ldots, f_{n'}')$ of $f_1', \ldots, f_{n'}'$. Define a relation R as follows:*

$$R = \{(\mathcal{C}(f_1, \ldots, f_n),\ \mathcal{C}'(f_1', \ldots, f_{n'}')) \mid f_1, \ldots, f_n, f_1', \ldots, f_{n'}' \in \mathcal{F}\}$$

Then in order to prove $f \approx f'$ it suffices to demonstrate that

1. *whenever $f \xrightarrow{i\,,o} h$, then $f' \xrightarrow{i\,,o} h'$ and $(h, h') \in R$; and*
2. *whenever $f' \xrightarrow{i\,,o} h'$, then $f \xrightarrow{i\,,o} h$ and $(h, h') \in R$.*

Proof. See [140]. ■

4.8.2 Verifying the equational laws

Checking that each equation in Proposition 4.2 holds in the semantics is a routine task. Here we only demonstrate (E13) to illustrate the use of the technique described above.

First, assume that

$$f[c;\, y \to x][g] \xrightarrow{i\,,o} f_1\ \ ,$$

for appropriate maps i and o. Then there exists a function block term f' and an output map o' such that $f_1 \overset{\text{def}}{=} f'[o'(\mathsf{y});\, \mathsf{y} \to \mathsf{x}][g]$, $o = (o')^g$ and the following is a valid derivation in the semantics:

$$\cfrac{\cfrac{f \xrightarrow{i^g + (c/\mathsf{x})\,,o'} f'}{f[c;\, \mathsf{y} \to \mathsf{x}] \xrightarrow{i^g\,,o'} f'[o'(\mathsf{y});\, \mathsf{y} \to \mathsf{x}]}\ (4.6)}{f[c;\, \mathsf{y} \to \mathsf{x}][g] \xrightarrow{i\,,o} f'[o'(\mathsf{y});\, \mathsf{y} \to \mathsf{x}][g]}\ (4.5)$$

Letting $g' = (g, \mathsf{z}/\mathsf{x})$ observe that, since z does not belong to the image of in$f$ under g,

$$(i + (c/\mathsf{z}))^{g'} = i^g + (c/\mathsf{x})$$

and thus

$$f \xrightarrow{(i+(c/\mathsf{z}))^{g'}\,,\,o'} f'$$

is exactly the same as:

$$f \xrightarrow{i^{g}+(c/\mathsf{x})\,,\,o'} f' \enspace .$$

Since, by definition of o, $o = (o')^{g} = (o')^{g'}$ and $o(g(\mathsf{y})) = o'(\mathsf{y})$ we can now construct the following valid derivation:

$$\cfrac{\cfrac{f \xrightarrow{(i+(c/\mathsf{z}))^{g'}\,,\,o'} f'}{f[g'] \xrightarrow{i+(c/\mathsf{z})\,,\,o} f'[g']}\;(4.5)}{f[g,\mathsf{z}/\mathsf{x}][c;\, g(\mathsf{y}) \to \mathsf{z}] \xrightarrow{i\,,\,o} f'[g,\mathsf{z}/\mathsf{x}][o'(\mathsf{y});\, g(\mathsf{y}) \to \mathsf{z}]}\;(4.6)$$

Thus, we have shown part 1 of Lemma 4.5. A completely symmetrical argument establishes the opposite direction, part 2. So, the relation consisting of all pairs of terms

$$f[c;\, \mathsf{y} \to \mathsf{x}][g], \quad f[g,\mathsf{z}/\mathsf{x}][c;\, g(\mathsf{y}) \to \mathsf{z}] \enspace ,$$

where $f \in \mathcal{F}$ and $\mathsf{z} \notin \operatorname{ran} g_0$, satisfies the criterion of Lemma 4.5. Consequently,

$$f[c;\, \mathsf{y} \to \mathsf{x}][g] \approx f[g,\mathsf{z}/\mathsf{x}][c;\, g(\mathsf{y}) \to \mathsf{z}]$$

as required.

4.9 Discussion

Recently, there have been a number of approaches to the formalisation of PLC programming languages, e.g. [36, 54, 55, 84]. These have focused on using formal definition and verification methods to support reasoning about the textual representations of these languages. However, an essential aspect of PLC languages is the ability to program using diagrams. This ability has an impact on the acceptance of the programming notation and on the designer's ability to reason (formally and informally) about the system.

Here we have established a precise correspondence between diagrams and the semantics of the language. We believe that this has significance both specifically for the programming of PLCs in such notation and more generally it establishes a rigorous criterion for the use of diagrammatic notations in domain-specific programming languages. In particular, the criterion regards whether a language featuring diagrammatic representation is sufficiently well designed to:

- ensure that all ways of constructing the same diagram are interpreted identically; and, thus,

- relieve programmers, through the use of diagrams, from the need to consider non-significant variants of programs.

It is interesting to note, in passing, how our work relates to some of the guidelines on domain-specific language selection from [72]; a set of industry-wide guidelines for the safe deployment of programmable controllers. Under the heading of "Closeness to application domain" (p. 150) [72] suggests that:

> *If an established representation of the application domain exists (e.g. textual representations such as mathematical symbols and equations, or graphical representations such as electronic circuit diagrams etc.) then the language syntax should be similar to this representation, and the semantics should be consistent with this syntax.*

Moreover:

> ... *there should be no difference between the coder's expectation and the compiler's interpretation.*

In function blocks one has both textual syntax, derived from data-flow equations, and diagrammatic syntax. Consistency is therefore required both across the two different representations and with respect to semantics. Thus, our work is this chapter provides a formalised, precisely stated criterion which wholly captures the essence of the above guidelines.

Chapter 5

Graphs and Categories

This chapter provides a self-contained introduction to those elements of the theory of categories which will be required in the remainder of this thesis. As graph-based notations form a wide class of visual and diagrammatic languages, particular attention is paid here to the category of graphs and to constructions within it. The material of section 5.5.2 in particular is motivated entirely by the needs of this thesis; the definitions and proofs therein are typically not found in standard texts (or, at least, those known to the author).

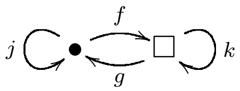
An often overwhelming feature of discourse in category theory is its reliance on an extensive stock of definitions; many of which describe generalisations of familiar concepts using new and sometimes alien terminology. Here, we have attempted to arrive at the concepts necessary for our purposes by using as few definitions as possible. (For instance, our account of monoidal categories in Section 5.6 makes no mention of either "bifunctors" or "natural transformations".) Consequently, we make no claim of this introduction being comprehensive for any purpose other than understanding the material in this thesis.

For thorough introductions to the subject the reader is referred to the books by Barr and Wells [8], and Pierce [114] which are targeted specifically for audiences in computer science. Shorter introductions, together with further pointers to the numerous applications of categories in computing, are provided in [115, 40, 119, 26]. The definitive text on category theory is MacLane's book [88].

5.1 Graphs

A *graph* consists of a set of objects O (vertices), a set of arrows A (edges) and a pair of functions $\partial_0, \partial_1 : A \to O$, called the *source* and *target* functions respectively. It is customary to write $a : u \to w$ to denote an arrow $a \in A$ such that $\partial_0(a) = u$ and $\partial_1(a) = w$.

Example 5.1. The graph G which is pictorially represented as



has $\{\bullet, \square\}$ as its set of objects and $\{f, g, j, k\}$ as its set of arrows. For f, $\partial_0(f) = \bullet$ and $\partial_1(f) = \square$. ■

Let the *sources* of a graph $G = (O, A, \partial_0, \partial_1)$ be those objects of G having no incident arrows:

$$\mathrm{Src}G \overset{\mathrm{def}}{=} \{o \mid \nexists a \in A.\ \partial_1(a) = o\} \ .$$

The *sinks* of G are those objects having no emanating arrows:

$$\mathrm{Snk}G \overset{\mathrm{def}}{=} \{o \mid \nexists a \in A.\ \partial_0(a) = o\} \ .$$

5.1.1 Graph homomorphisms

Intuitively, a homomorphism of graphs G and G' is a mapping h from the objects and arrows of G to the objects and arrows of G' such that whenever $a : u \to w$ is an arrow in G then $h(a) : h(u) \to h(w)$ in an arrow in G'. More precisely:

Definition 5.1. Let $G = (O, A, \partial_0, \partial_1)$ and $G' = (O', A', \partial'_0, \partial'_1)$ be graphs. A *homomorphism* $h : G \to G'$ is a pair $h = \langle h_O, h_A \rangle$ of functions $h_O : O \to O'$ and $h_A : A \to A'$ such that for all $a \in A$,

$$h_O(\partial_0(a)) = \partial'_0(h_A(a)) \text{ and } h_O(\partial_1(a)) = \partial'_1(h_A(a)) \ .$$

■

To simplify notation, we shall often denote both components h_O, h_A of a graph homomorphism h as simply h.

5.1.2 Reflexive graphs

A graph G is called *reflexive* if to every object u of G there is associated a distinguished arrow $u \to u$, called the *identity* arrow at u. Formally, a reflexive graph is a pair $\langle G, \mathrm{id} \rangle$, where $G = (O, A, \partial_0, \partial_1)$ is a graph and $\mathrm{id} : O \to A$ is a function such that $\partial_0(\mathrm{id}(u)) = \partial_1(\mathrm{id}(u)) = u$ for all $u \in O$.

Clearly, any graph G may be made reflexive by adding a distinguished arrow $\mathrm{id}(u) = 1_u : u \to u$ for each object u of G. The resulting graph is called the *reflexive closure* of G and is denoted $\mathcal{R}(G)$.

5.2 Categories

Any two arrows g, f in a graph such that the source of g coincides with the target of f are called *composable*. Thus, the set of all pairs of composable arrows in a graph $G = (O, A, \partial_0, \partial_1)$ is:

$$\mathrm{Comp}(G) \overset{\mathrm{def}}{=} \{\langle g, f\rangle \in A \times A \mid \partial_1(f) = \partial_0(g)\} \ .$$

A category is a reflexive graph in which every composable pair of arrows $f : A \to B$, $g : B \to C$ determines a specified *composite arrow* $g \circ f : A \to C$ subject to axioms.

Definition 5.2. A *category* $\mathcal{C}$ is a graph $(O, A, \partial_0, \partial_1)$ with two additional functions $\mathrm{id} : O \to A$ and $- \circ - : \mathrm{Comp}(\mathcal{C}) \to A$, called *identity* and *composition*, subject to the following axioms:

1. $\partial_0(\mathrm{id}(u)) = \partial_1(\mathrm{id}(u)) = u$
2. $\partial_0(g \circ f) = \partial_0(f)$ and $\partial_1(g \circ f) = \partial_1(g)$
3. $\mathrm{id}(w) \circ f = f$ and $f \circ \mathrm{id}(u) = f$, for $f : u \to w$
4. $k \circ (g \circ f) = (k \circ g) \circ f \quad .$

We shall usually write id_u (or 1_u, or just u) for $\mathrm{id}(u)$. ■

Thus, composition in a category is associative and has the arrows $\mathrm{id}(u)$ as identities.

5.2.1 Examples of categories

A great variety of mathematical structures may be regarded as the objects or arrows of appropriate categories. Here we present examples of such categories which will be of particular relevance to the rest of the present thesis.

5.2.1.1 The category of sets and functions

We assume that all sets mentioned hereafter belong to a sufficiently large universe (i.e. set of sets) U.

Let **Set** be the graph having:

- U as its set of objects (i.e. the objects of **Set** are all sets in U)
- arrows all (total) functions $f : A \to B$ between sets in U.

For each object A of **Set**, the identity arrow 1_A on A is the identity function on the set A: $1_A(x) \overset{\text{def}}{=} x$ for all $x \in A$. Composition of arrows in **Set** is the usual composition of functions: for arrows $f : A \to B$ and $g : B \to C$, $g \circ f$ is the function $A \to C$ defined by:

$$(g \circ f)(x) \overset{\text{def}}{=} g(f(x)), \text{ for all } x \in A \ .$$

It is well known that identity functions and composite functions satisfy the axioms in Definition 5.2. Thus **Set** is the category of all sets (in U) and functions between them.

Remark. *The restriction of all sets considered to the members of a universe U is required in order to avoid foundational paradoxes similar to Russell's paradox for set theory. In particular U is, by definition, not an object of* **Set**. *See [88], pages 21–24, for a thorough discussion on this topic.*

5.2.1.2 The category of graphs and graph homomorphisms

Let **Graph** be the graph having:

- objects: all graphs $(O, A, \partial_0, \partial_1)$ such that $O, A \in U$
- arrows: all graph homomorphisms $h : G \to G'$.

Graph may easily be seen to be a category. The identity arrow corresponding to a graph $G = (O, A, \partial_0, \partial_1)$ is the identity homomorphism $1_G = \langle 1_O, 1_A \rangle : G \to G$ (where 1_O and 1_A are the identity functions on O and A respectively). The composition of graph homomorphisms $h : G \to G'$ and $g : G' \to G''$ is defined in terms of function composition:

$$g \circ h \overset{\text{def}}{=} \langle g_O \circ h_O, g_A \circ h_A \rangle : G \to G'' \ .$$

5.2.1.3 Sequences and Permutations

For every $n \in \mathbb{N}$, let $[n]$ abbreviate $\{1, \dots, n\}$. Given any set A, a *finite sequence* over A is a function $s : [n] \to A$ for some $n \in \mathbb{N}$.

A bijection[1] of the form $p : [n] \to [n]$ is called a *permutation* of $\{1, \dots, n\}$. Permutations may be regarded as the arrows of a simple category **Perm** having:

- $\mathbb{N}$ as its set of objects;

[1] A bijection is a function which is both injective (i.e. "one-to-one") and surjective (i.e. "onto").

- arrows $p : n \to n$ all permutations $p : [n] \to [n]$;

- composition that of functions with the obvious identities $1_k : k \to k$, where $1_k(x) \overset{\text{def}}{=} x$ for all $1 \leq x \leq k$.

In particular, note that **Perm** contains no arrows $a : m \to n$ for $m \neq n$.

5.2.2 Commutative diagrams

A *diagram in a category* $\mathcal{C}$ is a graph D whose objects and arrows are consistently labelled with objects and arrows in $\mathcal{C}$: that is, whenever an arrow a of D is labelled f and $\partial_0(a)$, $\partial_1(a)$ are labelled A and B respectively, then $f : A \to B$ is an arrow in $\mathcal{C}$. (Thus, formally, a diagram in $\mathcal{C}$ is a homomorphism from a graph D to the graph underlying $\mathcal{C}$.)

Example 5.2. Let $f : A \to B$, $g : B \to B$ be functions. Then,

$$\begin{tikzcd}
A \arrow[r, shift left, "f"] \arrow[r, shift right, "f"'] & B \arrow[loop right, "g"] \arrow[d, "1_B"] \\
& B
\end{tikzcd}$$

is a diagram in **Set**. ■

A diagram D in a category $\mathcal{C}$ is said to *commute* (or to be *commutative*) if, for every pair of objects A, B, all the paths in D from A to B determine equal arrows in $\mathcal{C}$. Thus, to say that the diagram

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "h"'] & B \arrow[d, "g"] \\
C \arrow[r, "k"'] & D
\end{tikzcd}$$

commutes is tantamount to asserting that $g \circ f = k \circ h$. (Readers interested in a formal definition of commutativity are advised to consult chapter 4 of [8].)

5.3 Isomorphisms

The reader may recall from set theory that two sets A, B are considered *isomorphic* if there is a bijection $f : A \to B$. Equivalently, A and B are isomorphic if there exist functions $f : A \to B$ and $g : B \to A$ such that $g \circ f = 1_A$ and $f \circ g = 1_B$. The concept of isomorphic sets admits an immediate generalisation in terms of arrows in an arbitrary category:

Definition 5.3. An arrow $m : A \to B$ in a category $\mathcal{C}$ is called an *isomorphism* if there exists arrow $m^{-1} : B \to A$ such that $m \circ m^{-1} = 1_B$ and $m^{-1} \circ m = 1_A$. If such a pair of arrows exists between A and B, then A is called *isomorphic to* B, written $A \cong B$. ■

Thus, in **Set** two objects are isomorphic if they are so in the usual set-theoretic sense. In **Graph**, $G \cong G'$ if there exist graph homomorphisms $h : G \to G'$ and $g : G' \to G$ satisfying the condition of Definition 5.3 (or, equivalently, if both components of h are bijections).

Lemma 5.1. *Let $i : A \to B$ be an isomorphism in a category $\mathcal{C}$. For any two arrows $f, g : B \to C$ in $\mathcal{C}$, $f \circ i = g \circ i$ implies $f = g$.*

Proof. Let $j : B \to A$ be the arrow such that $i \circ j = 1_B$ and $j \circ i = 1_A$ and assume $f \circ i = g \circ i$. Then, $f = f \circ 1_B = f \circ (i \circ j) = (f \circ i) \circ j = (g \circ i) \circ j = g \circ 1_B = g$. ■

It is straightforward to verify that $\cong$ is an equivalence relation (i.e. one that is reflexive, symmetric and transitive) on the objects of any category.

Many other familiar set-theoretic concepts (among them that of subset, disjoint union and cartesian product) have category-theoretic generalisations. In the following two sections we consider two such generalisations which will be required in Chapter 7.

5.4 Coproducts

Definition 5.4. Let A, B be objects in a category $\mathcal{C}$. A *coproduct* of A, B is an object $A + B$ together with two arrows $\mathrm{inl}_{A,B} : A \to A + B$ and $\mathrm{inr}_{A,B} : B \to A + B$ such that, for any other object C and arrows $h : A \to C$, $k : B \to C$, there exists a unique arrow $[h, k] : A + B \to C$ such that $[h, k] \circ \mathrm{inl}_{A,B} = h$ and $[h, k] \circ \mathrm{inr}_{A,B} = k$. That is, both triangles in the following diagram commute:

$$\begin{tikzcd}
A \arrow[r, "\mathrm{inl}_{A,B}"] \arrow[dr, "h"'] & A + B \arrow[d, "{[h,k]}"] & B \arrow[l, "\mathrm{inr}_{A,B}"'] \arrow[dl, "k"] \\
 & C &
\end{tikzcd}$$

■

A category $\mathcal{C}$ is said to have coproducts if every two objects in $\mathcal{C}$ have a coproduct.

Example 5.3. In **Set**, the *disjoint union* $A \uplus B \overset{\text{def}}{=} (A \times \{0\}) \cup (B \times \{1\})$ is a coproduct of sets A and B. (Here, × denotes the cartesian product of sets.) In this case the injections $\text{inl}_{A,B} : A \to A + B$, $\text{inr}_{A,B} : B \to A + B$ are the mappings: $a \in A \overset{\text{inl}}{\mapsto} \langle a, 0\rangle$ and $b \in B \overset{\text{inr}}{\mapsto} \langle b, 1\rangle$. ■

Coproducts extend to arrows. Let $f : A \to A'$, $g : B \to B'$. Provided that both $A + A'$ and $B + B'$ exist, the arrow $f + g$ is the *unique* arrow $h : A + B \to A' + B'$ such that $h \circ \text{inl}_{A,A'} = \text{inl}_{B,B'} \circ f$ and $h \circ \text{inr}_{A,A'} = \text{inr}_{B,B'} \circ g$. (That is, $f + g = [\text{inl}_{A',B'} \circ f, \text{inr}_{A',B'} \circ g]$.)

Lemma 5.2. *In any category with coproducts,* $(g+g')\circ(f+f') = (g\circ f)+(g'\circ f')$.

Proof. Let $f : A \to B$, $g : B \to C$, $f' : A' \to B'$ and $g' : B' \to C'$. By the defining properties of $f + f'$ and $g + g'$, all four inner squares in

$$\begin{tikzcd}
A \arrow[r, "\text{inl}"] \arrow[d, "f"'] & A + A' \arrow[d, "f+f'"] & A' \arrow[l, "\text{inr}"'] \arrow[d, "f'"] \\
B \arrow[r, "\text{inl}"] \arrow[d, "g"'] & B + B' \arrow[d, "g+g'"] & B' \arrow[l, "\text{inr}"'] \arrow[d, "g'"] \\
C \arrow[r, "\text{inl}"] & C + C' & C' \arrow[l, "\text{inr}"']
\end{tikzcd}$$

commute. It follows that

$$((g + g') \circ (f + f')) \circ \text{inl}_{A,A'} = \text{inl}_{C,C'} \circ (g \circ f)$$

and

$$((g + g') \circ (f + f')) \circ \text{inr}_{A,A'} = \text{inr}_{C,C'} \circ (g' \circ f') \ .$$

Thus, $(g+g')\circ(f+f')$ satisfies the property with respect to which $(g\circ f)+(g'\circ f')$ is unique. Hence, $(g \circ f) + (g' \circ f') = (g + g') \circ (f + f')$. ■

Coproducts are uniquely defined only up to isomorphism: if both C and D are coproducts of A and B then there is *unique* isomorphism $C \cong D$.

Often, one is interested only in a specific coproduct for any two objects in a category $\mathcal{C}$. In that case, the chosen coproduct is denoted $A + B$ and is referred to as just "*the* coproduct" in $\mathcal{C}$. For instance, the coproduct in **Set** is that given by disjoint union. Here is a further examples that we shall use later:

5.4.1 The coproduct in Graph

Let $G = (O, A, \partial_0, \partial_1)$, $G' = (O', A', \partial'_0, \partial'_1)$ be graphs and let

$$G + G' \overset{\mathrm{def}}{=} (O + O',\ A + A',\ \partial_0 + \partial'_0,\ \partial_1 + \partial'_1)$$

(on the right-hand side of $\overset{\mathrm{def}}{=}$, the symbol $+$ denotes the coproduct of objects and arrows in **Set**). The graph $G + G'$ is the coproduct of graphs G and G' in the category **Graph**. In the usual pictorial representation of graphs, $G + G'$ would result in the juxtaposition of two disjoint copies of the representations of G and G'.

5.5 Pushouts

Let $E \subseteq A \times A$ be an equivalence relation on a set A. For any $a \in A$, the *equivalence class* of a under E, denoted $[a]_E$, is the set $\{a' \in A \mid \langle a, a' \rangle \in E\}$ of all elements equivalent to a. The set

$$\{[a]_E \mid a \in A\}$$

of all such equivalence classes is called the *quotient set* of A by E and denoted A/E. The function $q : A \to A/E$ taking each $a \in A$ to its equivalence class $[a]_E$ is called the *quotient map* induced by E.

Definition 5.5 (Amalgam of Sets). Let A, B, C be sets and $f : C \to A$, $g : C \to B$ be functions with common domain C. The *amalgam* of A and B through f and g, denoted $A \amalg_{f,g} B$, is the set obtained from $A + B$ via the quotient map induced by making the identifications

$$\mathrm{inl}(f(x)) = \mathrm{inr}(g(x)),\ \ x \in C\ \ .$$

■

(I.e., the quotient map in question is induced by the least equivalence relation E on $A + B$ such that $\langle \mathrm{inl}(f(x)), \mathrm{inr}(g(x)) \rangle \in E$ for all $x \in C$.)

In words, the amalgamated sum $A \amalg_{f,g} B$ identifies the elements $f(x)$ and $g(x)$ in the coproduct of A and B. The following is a known property of $\amalg$:

Lemma 5.3. *Let D be any set and $d_A : A \to D$, $d_B : B \to D$ be functions such that $d_A \circ f = d_B \circ g$, where $f : C \to A$, $g : C \to B$. Then:*

1. *$q_A \circ f = q_B \circ g$, where $q_A \overset{\mathrm{def}}{=} q \circ \mathrm{inl}_{A,B}$, $q_B \overset{\mathrm{def}}{=} q \circ \mathrm{inr}_{A,B}$, and q is the quotient map of Definition 5.5;*

2. *there is unique $h : A \amalg_{f,g} B \to D$ such that $d_A = h \circ q_A$ and $d_B = h \circ q_B$:*

$$\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "g"'] & A \arrow[d, "q_A"'] \arrow[ddr, bend left, "d_A"] & \\
B \arrow[r, "q_B"] \arrow[drr, bend right, "d_B"'] & A \amalg_{f,g} B \arrow[dr, "h"] & \\
& & D
\end{tikzcd}$$

Proof. Part 1 obviously holds. For part 2, observe that the requirements $d_A = h \circ q_A$ and $d_B = h \circ q_B$ define h uniquely. For suppose $h' : A \amalg_{f,g} B \to D$ is another function such that $d_A = h' \circ q_A$ and $d_B = h' \circ q_B$. Since $d_A \circ f = d_B \circ g$ one also has $h \circ (q_A \circ f) = h' \circ (q_B \circ g)$. By $q_A \circ f = q_B \circ g$ and the fact that q_A, q_B are surjective it follows that $h = h'$. ■

Pushouts are the category-theoretic generalisation of amalgams in **Set**. In the general setting, one starts with two arrows $f : C \to A$ and $g : C \to B$ in a category $\mathcal{C}$ having common source C. The idea is that a pushout construction "combines" the objects A and B into a third object P by making as few identifications as possible (those "specified" by f and g) and by adding nothing which is essentially new [40]. The resulting object P comes equipped with arrows $p_1 : A \to P$ and $p_2 : B \to P$ showing how A and B are "included" in P.

Definition 5.6. A commutative square

$$\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "g"] & A \arrow[d, "p_1"'] \\
B \arrow[r, "p_2"] & P
\end{tikzcd}$$

in a category is called a *pushout square* if whenever there is object Q and arrows $q_1 : A \to Q$, $q_2 : B \to Q$ such that $q_1 \circ f = q_2 \circ g$, there is a unique arrow $h : P \to Q$ such that $q_i = h \circ p_i$, $i = 1, 2$. That is, both triangles in the following diagram commute:

$$\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "g"'] & A \arrow[d, "p_1"'] \arrow[ddr, bend left, "q_1"] & \\
B \arrow[r, "p_2"] \arrow[drr, bend right, "q_2"'] & P \arrow[dr, "h"] & \\
& & Q
\end{tikzcd}$$

The triple (P, p_1, p_2) is called a *pushout* of f, g. ■

A category in which a pushout exists for every pair of arrows $f : A \to B$ and $g : A \to C$ in the category is said to have pushouts.

5.5.1 Properties of pushouts

Like coproducts, pushouts are only defined up to isomorphism:

Proposition 5.1. *Let both (P, p_A, p_B) and (P', p'_A, p'_B) be pushouts of the same pair $f : C \to A$, $g : C \to B$ of arrows. Then, there exists unique isomorphism $i : P \to P'$.*

Pushout squares "compose" in a particular way:

Lemma 5.4. *If the inner squares in* $\begin{tikzcd} \cdot \arrow[r] \arrow[d] & \cdot \arrow[r] \arrow[d] & \cdot \arrow[d] \\ \cdot \arrow[r] & \cdot \arrow[r] & \cdot \end{tikzcd}$ *are pushout squares, then so is the outer rectangle.* ■

When coproducts exist, any two pushout squares give rise to a third:

Lemma 5.5. *In category with coproducts, if the two leftmost squares in*

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "h"] \\ C \arrow[r, "k"'] & D \end{tikzcd} \qquad \begin{tikzcd} A' \arrow[r, "f'"] \arrow[d, "g'"'] & B' \arrow[d, "h'"] \\ C' \arrow[r, "k'"'] & D' \end{tikzcd} \qquad \begin{tikzcd} A + A' \arrow[r, "f+f'"] \arrow[d, "g+g'"'] & B + B' \arrow[d, "h+h'"] \\ C + C' \arrow[r, "k+k'"'] & D + D' \end{tikzcd}$$

are pushouts, then so is the third. ■

5.5.2 Gluing graphs

Pushouts provide a general framework for combining (or "gluing") two graphs by identifying some of their objects and arrows. In particular, variants of pushout constructions have been used extensively as algebraic tools in the theory of graph transformations and graph grammars (see, e.g. [28, 24]). Here we present the details of the pushout construction in **Graph**, which is induced by amalgams of sets.

Definition 5.7. Let $G = (O, A, \partial_0, \partial_1)$, $G' = (O', A', \partial'_0, \partial'_1)$ be graphs and $h : B \to G$, $h' : B \to G'$ be graph homomorphisms. The graph

$$G \amalg_{h,h'} G' = (O'', A'', \partial''_0, \partial''_1)$$

has:

- $O'' = O \amalg_{h,h'} O'$ (the amalgam of sets O, O' along the object parts of h, h') as its set of objects;

- $A'' = A \amalg_{h,h'} A'$ (the amalgam of sets A, A' along the arrow parts of h, h') as its set of arrows;

- ∂''_0 defined by $\partial''_0 \circ q = q' \circ (\partial_0 + \partial'_0)$, where q and q' are the quotient maps associated with $O \amalg_{h,h'} O'$ and $A \amalg_{h,h'} A'$ respectively, as in Definition 5.5;

- ∂''_1 defined by $\partial''_1 \circ q = q' \circ (\partial_0 + \partial'_0)$. ■

Proposition 5.2. *Let G, G', h and h' be as in Definition 5.7. There are graph homomorphisms s, s' such that*

$$\begin{tikzcd}
B \arrow[r, "h"] \arrow[d, "h'"'] & G \arrow[d, "s"] \\
G' \arrow[r, "s'"'] & G \amalg_{h,h'} G'
\end{tikzcd}$$

is a pushout square in **Graph**.

Proof. Take $s \overset{\text{def}}{=} \langle q \circ \text{inl}_{O,O'}, q' \circ \text{inl}_{A,A'} \rangle$ and $s' \overset{\text{def}}{=} \langle q \circ \text{inr}_{O,O'}, q' \circ \text{inr}_{A,A'} \rangle$ where q, q' are as in Definition 5.7. Clearly, s and s' are graph homomorphisms. Moreover they can be easily shown to satisfy $s \circ h = s' \circ h'$.

Let $B = (B_O, B_A, \dots)$. Assume now that $T = (T_O, T_A, \dots)$ is a graph and that $t : G \to T$, $t' : G' \to T$ are homomorphisms satisfying $t \circ h = t' \circ h'$. Let $u_O : O \amalg_{h,h'} O' \to T_O$ and $u_A : A \amalg_{h,h'} A' \to T_A$ be the unique functions asserted by Lemma 5.3 applied to the following situations:

$$\begin{tikzcd}
B_O \arrow[r, "h"] \arrow[d, "h'"'] & O \arrow[d, "s"] \arrow[ddr, bend left, "t"] & \\
O' \arrow[r, "s'"] \arrow[drr, bend right, "t'"'] & O \amalg_{h,h'} O' \arrow[dr, "u_O"] & \\
 & & T_O
\end{tikzcd}
\qquad
\begin{tikzcd}
B_A \arrow[r, "h"] \arrow[d, "h'"'] & A \arrow[d, "s"] \arrow[ddr, bend left, "t"] & \\
A' \arrow[r, "s'"] \arrow[drr, bend right, "t'"'] & A \amalg_{h,h'} O' \arrow[dr, "u_A"] & \\
 & & T_A
\end{tikzcd}$$

Thus, $u = \langle u_O, u_A \rangle$ is the unique homomorphism $u : G \amalg_{h,h'} G' \to T$ such that $t = u \circ s$ and $t' = u \circ s'$. That is, the square in the statement of the proposition is a pushout square. ■

5.6 Tensor categories

In this section we introduce a family of categories which, in addition to the binary composition of arrows, carry extra algebraic structure. This structure comes in the form of a binary operator on both the objects and arrows of the category.

This operator is generically denoted $\otimes$ and called *tensor* or *monoidal product.* (The name "monoidal product" derives from the fact that the restriction of $\otimes$ to the objects of the category forms a monoid.)

Let $\mathcal{C}$ be a category with objects O and arrows A, and $\mathcal{B}$ be a pair $\langle \mathcal{B}_O, \mathcal{B}_A \rangle$ of functions $\mathcal{B}_O : O \times O \to O$ and $\mathcal{B}_A : A \times A \to A$. We shall say that $\mathcal{B}$ is a *binary operator* on $\mathcal{C}$ if, whenever $f : A \to B$ and $f' : A' \to B'$ are arrows of $\mathcal{C}$, then

$$\mathcal{B}_A(f, f') : \mathcal{B}_O(A, A') \to \mathcal{B}_O(B, B') \text{ in } \mathcal{C} \ .$$

When the distinction is made clear from the context, we shall use $\mathcal{B}$ to denote both the object and arrow components of a binary operator $\mathcal{B}$. The application of binary operators will be written using infix notation (e.g. $f \mathcal{B} g$ instead of $\mathcal{B}(f, g)$).

Example 5.4. A binary operator "$\otimes$" on **Perm** (Section 5.2.1.3) may be defined as follows:

- for any two objects n, m of **Perm**, $n \otimes m \overset{\text{def}}{=} n + m$;
- for any two arrows $p : n \to n$ and $p' : m \to m$ let $p \otimes p' : n + m \to n + m$ be the permutation:

$$(p \otimes p')(x) \overset{\text{def}}{=} \begin{cases} p(x), & 1 \leq x \leq n \\ p'(x - n) + n, & n < x \leq n + m \end{cases} \qquad \blacksquare$$

In the context of tensor categories it is customary to denote each composite arrow $g \circ f$ as $f; g$. ($f; g$ is often referred to as the composition of f and g in *diagrammatic order.*)

Definition 5.8. A *tensor category* is a triple $(\mathcal{C}, \otimes, e)$ consisting of

- a category $\mathcal{C}$
- a binary operator $\otimes$ on $\mathcal{C}$; and
- an object e of $\mathcal{C}$

subject to the following axioms:

1. $A \otimes e = e \otimes A = A$, for all objects A of $\mathcal{C}$
2. $(A \otimes B) \otimes C = A \otimes (B \otimes C)$, for all objects A, B, C in $\mathcal{C}$
3. $1_A \otimes 1_B = 1_{A \otimes B}$ for all objects A, B in $\mathcal{C}$
4. $f \otimes 1_e = 1_e \otimes f = f$, for all arrows f in $\mathcal{C}$.

5. $(f \otimes g) \otimes h = f \otimes (g \otimes h)$ for all arrows f, g and h in $\mathcal{C}$

6. $(f; g) \otimes (f'; g') = (f \otimes f'); (g \otimes g')$, for all arrows $f : A \to B$, $g : B \to C$, $f' : A' \to B'$ and $g' : B' \to C'$ in $\mathcal{C}$. ■

Remark. *What we have called a tensor category is usually referred to in the literature as a* strict monoidal category *(e.g. [88], page 137). In so-called non-strict monoidal categories, axioms 1, 2 and 4, 5 above are only required to hold up to isomorphism: e.g., one has $A \otimes e \cong A$ instead of $A \otimes e = A$. Moreover, the mediating isomorphisms are required to satisfy extra coherence conditions.*

Example 5.5. Let $\otimes$ be as in Example 5.4. $(\mathbf{Perm}, \otimes, 0)$ is a tensor category. Clearly, $n \otimes 0 = 0 \otimes n = n$ and $n \otimes (m \otimes k) = (n \otimes m) \otimes k$. Axioms 3, 4 and 5 are also easy to establish: $1_n \otimes 1_m = 1_{n+m}$, $(p \otimes 1_0) = p = (1_0 \otimes p)$ and $p \otimes (p' \otimes p'') = (p \otimes p') \otimes p''$. To establish the last axiom, consider $p, p' : n \to n$, $s, s' : m \to m$ and let $\pi = (p \otimes s); (p' \otimes s')$ and $\pi' = (p; p') \otimes (s; s')$. For $1 \leq x \leq n$, $\pi(x) = p'(p(x)) = (p; p')(x) = \pi'(x)$. For $n+1 \leq x \leq n+m$, $\pi(x) = (p' \otimes s')((p \otimes s)(x)) = (p' \otimes s')(s(x-n)+n) = s'(s(x-n)) + n = (s; s')(x-n) + n = \pi'(x)$. Thus $\pi(x) = \pi'(x)$ for all $1 \leq x \leq n+m$. ■

5.6.1 Symmetric tensor categories

The reader may have noticed that the tensor in a tensor category is not required to be commutative, i.e. $A \otimes B = B \otimes A$ does not hold in general. In many situations, however, there is a natural isomorphism between $A \otimes B$ and $B \otimes A$ for all pairs of objects A, B in the category. This isomorphism is captured as an arrow $\mathbf{c}_{A,B} : A \otimes B \to B \otimes A$, called the *symmetry* on A and B, satisfying certain intuitive conditions.

Definition 5.9. A *symmetry* for a tensor category $(\mathcal{C}, \otimes, e)$ is a family

$$\{\mathbf{c}_{A,B} : A \otimes B \to B \otimes A \mid A, B \text{ are objects in } \mathcal{C}\}$$

of isomorphisms (in the sense of Definition 5.3) such that the following conditions hold:

1. $\mathbf{c}_{A,B} \,;\mathbf{c}_{B,A} = 1_{A \otimes B}$

2. $\mathbf{c}_{A \otimes B, C} = (1_A \otimes \mathbf{c}_{B,C}) \,; (\mathbf{c}_{A,C} \otimes 1_B)$

3. $(f \otimes g) \,; \mathbf{c}_{A',B'} = \mathbf{c}_{A,B} \,; (g \otimes f)$, for all arrows $f : A \to A'$ and $g : B \to B'$ in $\mathcal{C}$.

A tensor category equipped with a specified symmetry will be called a *symmetric tensor category.* ■

Example 5.6. Let $\pi_{n,m}$ be the permutation of $[n+m]$ defined as:

$$\pi_{n,m}(x) \overset{\text{def}}{=} \begin{cases} m+x, & 1 \leq x \leq n \\ x-n, & n < x \leq n+m \end{cases} .$$

We shall call each $\pi_{n,m}$ a *symmetric permutation.* Recalling that $n \otimes m \overset{\text{def}}{=} n+m$, the family $\{\pi_{n,m} \mid n, m \in \mathbb{N}\}$ is a symmetry for the tensor category $(\mathbf{Perm}, \otimes, 0)$:

1. $\pi_{n,m} \,;\, \pi_{m,n} = 1_{n+m}$

2. $\pi_{n+m,k} = (1_n \otimes \pi_{m,k}) \,;\, (\pi_{n,m} \otimes 1_k)$

3. For any two permutations $p : n \to n$ and $p' : m \to m$,

$$(p \otimes p') \,;\, \pi_{n,m} = \pi_{n,m} \,;\, (p' \otimes p) \ .$$

The first two equations should be obvious. The third is established by simple case analysis and appeal to the definition of $\otimes$ on permutations:

- For $1 \leq x \leq n$, $((p \otimes p'); \pi_{n,m})(x) = \pi_{n,m}(p(x)) = p(x) + m = p(m + x - m) + m = (p' \otimes p)(m + x) = (\pi_{n,m}; (p' \otimes p))(x)$.

- For $n < x \leq n+m$, $((p \otimes p'); \pi_{n,m})(x) = \pi_{n,m}(p'(x-n) + n) = p'(x-n) = (p' \otimes p)(x-n) = (\pi_{n,m}; (p' \otimes p))(x)$. ■

5.6.2 Traces on tensor categories

Another common feature of many symmetric tensor categories is what Joyal, Street and Verity have called a “trace” [75]; categories equipped with a trace being called “traced (tensor) categories”. The application of traced categories in computing is growing rapidly as many examples of iteration, feedback and general recursion operators have been shown to be instances of traces. These include feedback in the calculus of “flownomials” [127], feedback in asynchronous data-flow networks [68] and circuits [80], recursion from cyclic sharing [64], and reflection in action calculi [101] among many others.

Our interest in traces stems from the fact that their axioms neatly capture a particular notion of equivalence between diagrams containing “loops”. Here, we briefly introduce the axioms for traces and defer their diagrammatic presentation until the following chapter. The following definition is therefore provided only as reference; the reader should not be deterred by what might at first appear as an overwhelmingly complicated set of axioms. In particular, an appealing diagrammatic representation of the axioms exists (given in figures 6.6 and 6.7), which underpins the material in the next chapter.

Definition 5.10. Let $\mathcal{C}[A, B]$ denote the set of all arrows $a : A \to B$ in a category $\mathcal{C}$. Let also $\mathcal{T} = (\mathcal{C}, \otimes, e)$ be a symmetric tensor category with symmetry $\mathbf{c}$. A family Tr of functions

$$\mathrm{Tr}^X_{A,B} : \mathcal{C}[X \otimes A, X \otimes B] \longrightarrow \mathcal{C}[A, B]$$

is called a *trace* on $\mathcal{T}$ if it satisfies the following axioms:

1. Vanishing:
$$\mathrm{Tr}^e_{A,B}(f) = f : A \to B$$
where $f : A \longrightarrow B$, and
$$\mathrm{Tr}^{X\otimes Y}_{A,B}(f) = \mathrm{Tr}^Y_{A,B}(\mathrm{Tr}^X_{Y\otimes A,Y\otimes B}(f))$$
where $f : X \otimes Y \otimes A \longrightarrow X \otimes Y \otimes B$

2. Superposing:
$$\mathrm{Tr}^X_{A\otimes C,B\otimes C}(f \otimes 1_C) = \mathrm{Tr}^X_{A,B}(f) \otimes 1_C$$
where $f : X \otimes A \longrightarrow X \otimes B$

3. Yanking:
$$\mathrm{Tr}^X_{X,X}(c_{X,X}) = 1_X$$

4. Left Tightening:
$$\mathrm{Tr}^X_{A,B}((1_X \otimes g); f) = g; \mathrm{Tr}^X_{A',B}(f)$$
where $f : X \otimes A' \longrightarrow X \otimes B$ and $g : A \longrightarrow A'$

5. Right Tightening:
$$\mathrm{Tr}^X_{A,B}(f; (1_X \otimes g)) = \mathrm{Tr}^X_{A,B'}(f); g$$
where $f : X \otimes A \longrightarrow B' \otimes X$ and $g : B' \longrightarrow B$

6. Sliding:
$$\mathrm{Tr}^X_{A,B}(f; (1_B \otimes g)) = \mathrm{Tr}^Y_{A,B}((1_A \otimes g); f)$$
where $f : X \otimes A \longrightarrow Y \otimes B$ and $g : Y \longrightarrow X$. ■

Remark. *The above axiomatisation of traces in due to Hasegawa [63, 64] and is a specialised application of the general definition in [75] which applies to a wide class of* tortile tensor categories *[126].*

5.6.3 Units and counits

Beyond symmetries, symmetric tensor categories often come equipped with other special arrows. In later chapters we shall make use of two such special arrows, which we call "unit" and "counit" (pronounced "co-unit").

Definition 5.11. Let $(\mathcal{C}, \otimes, e)$ be a symmetric tensor category in which every arrow $f : A \to B$ has an associated "dual" arrow $f^\star : B \to A$. Given an object A of $\mathcal{C}$, a *unit-counit pair for* A is a pair $\langle \mu_A, \eta_A \rangle$ of arrows $\mu_A : e \to A \otimes A$ and $\eta_A : A \otimes A \to e$, such that:

1. $(\mu_A \otimes 1_A)\,;\,(1_A \otimes \eta_A) = 1_A$
2. $(1_A \otimes \mu_A)\,;\,(\eta_A \otimes 1_A) = 1_A$
3. $\mu_A\,;\,(f \otimes 1_A) = \mu_B\,;\,(1_B \otimes f^\star)$ for all arrows $f : A \to B$
4. $(f^\star \otimes 1_A)\,;\,\eta_A = (1_B \otimes f)\,;\,\eta_B$ for all arrows $f : A \to B$. ■

The last two axioms require that the following two diagrams commute:

$$\begin{tikzcd}
e \arrow[r, "\mu_A"] \arrow[d, "\mu_B"'] & A \otimes A \arrow[d, "f\otimes 1_A"] \\
B \otimes B \arrow[r, "1_B\otimes f^\star"'] & B \otimes A
\end{tikzcd}
\qquad
\begin{tikzcd}
B \otimes A \arrow[r, "f^\star\otimes 1_A"] \arrow[d, "1_B\otimes f"'] & A \otimes A \arrow[d, "\eta_A"] \\
B \otimes B \arrow[r, "\eta_B"'] & e
\end{tikzcd}$$

A symmetric tensor category is said to have units and counits if a unit-counit pair exists for every object in the category.

Remark. *In general, the function* $-^\star$ *is also defined on the objects of the category, giving for each object* A *its "dual"* $A^\star$*. Thus, what we have called "unit-counit pairs" correspond to the components of the unit and counit (di)natural transformations ([88], page 214) of a self-dual, compact-closed structure [81] on a symmetric monoidal category. Self-duality in this context means* $A^\star = A$ *for all objects* A *in the category.*

There is a standard way in which unit-counit pairs equip a symmetric tensor category with a canonical trace:

Proposition 5.3. *In any category with units* μ *and counits* η*, a trace is defined by the formula:*

$$\mathrm{Tr}^X_{A,B}(f) \overset{\text{def}}{=} (\mu_X \otimes 1_A)\,;\,(1_X \otimes f)\,;\,(\eta_X \otimes 1_B)\ .$$

Proof. Units and counits make the category an instance of a tortile tensor category [126]. The result then follows from Proposition 3.1 of [75]. Thus, the canonical trace of $f : X \otimes A \to X \otimes B$ is the composite arrow:

$$A \xrightarrow{\mu_X \otimes 1_A} X \otimes X \otimes A \xrightarrow{1_X \otimes f} X \otimes X \otimes B \xrightarrow{\eta_X \otimes 1_B} B \quad .$$

■

5.7 Further Remarks

Category theory requires a severe paradigm shift from a "sets with elements" view to that of "objects and arrows". In the study of diagrammatic and other representational systems, we believe that such a shift in perspective may be very well worth making. This is not least because typical mathematical descriptions of diagrams, exemplified by our Definition 4.3, almost immediately induce notions of "diagram (homo)morphism" (recall the remark following Definition 4.3). Such morphisms underpin notions of equivalence between diagrams (such as isomorphism) and may be further refined to capture situations of diagrams abstracting other diagrams etc. Thus, by taking such morphisms into account, one typically obtains not just a set of (models of) diagrams in the same class, but rather a category thereof. We expect that category-theoretic tools will become indispensable when representational systems, rather than individual diagrams, become one's object of study.

Chapter 6

Formalising Pragmatic Features

Graph-based notations form a significant subclass of visual languages, particularly in computer science and many engineering domains. Typical formalisations of such notations often pay scant regard to pragmatic considerations such as the spatial layout of such graphs. However, studies of the use of graph-based notations in practice have shown that users employ layout to capture important information concerning the semantics of the domain being represented. Furthermore, this extra information typically supports reasoning tasks over these graph-based representations, the structure of a proof being directed by the structure of the graph. Typical approaches to the formalisation of graphs generally do not capture such pragmatic considerations in a manner which would be sufficient to account for this kind of support for reasoning. Graph grammars for example (surveyed in [25]), generally treat spatial relations as uninterpreted, and thus in some sense as "meaningless". By contrast theories based upon spatial logics (see [95] for a survey of both these and grammatical approaches) do account for layout characteristics in diagrams, but their expressiveness typically far exceeds what is required for graph-based notations, making them too cumbersome for our purposes.

This chapter highlights an algebraic approach to the formalisation of graph-based notations which is sensitive to relevant layout information. In the following section we summarise the argument, supported by practical studies, for the necessity of including such pragmatic aspects in formalisations. Section 6.2 presents an example SFC diagram, taken from software engineering practice, which both motivates and illustrates the subsequent presentation. Sections 6.3 and 6.4 describe the salient aspects of graphs, and of our example language, which our algebra seeks to capture. The algebra itself is summarised in Section 6.4.1 and, in Section 6.4.2, provides a formalisation of our example. Finally, Sections 6.5 and 6.6 illustrate how this algebraic formalisation both captures pragmatic aspects of our example and how this in turn supports direct reasoning over its structure.

6.1 Visual language pragmatics

In linguistic theories of human communication, developed initially for written text or spoken dialogues, theories of "pragmatics" seek to explain how conventions and patterns of language use carry information over and above the literal truth value of sentences. For example, in the discourse:

1. (a) The lone ranger jumped on his horse and rode into the sunset.

   (b) The lone ranger rode into the sunset and jumped on his horse.

(1a)'s implicature is that the jump happened first, followed by the riding. By contrast, (1b)'s implicature is that riding preceded jumping. In both (1a) and (1b), implicatures go beyond the literal truth conditional meaning. For instance, all that matters for the truth of a complex sentence of the form P *and* Q is that both P and Q be true; the order of mention of the components is irrelevant. Pragmatics, thus, helps to bridge the gap between truth conditions and "real" meaning. This concept applies equally well to the use of visual languages in practice. Indeed, there is a recent history of work which draws parallels between pragmatic phenomena which occur in natural language, and for which there are established theories, and phenomena occurring in visual languages [47, 94, 110].

Studies of digital electronics engineers using CAD systems for designing the layout of computer circuits demonstrated that the most significant difference between novices and experts is in the use of layout to capture domain information [113]. In such circuit diagrams the layout of components is not specified as being semantically significant. Nevertheless, experienced designers exploit layout to carry important information by grouping together components which are functionally related. By contrast, certain diagrams produced by novices were considered poor because they either failed to use layout or, in particularly "awful" examples, were especially confusing through their mis-use of the common layout conventions adopted by the experienced engineers. The correct use of such conventions is thus seen as a significant characteristic distinguishing expert from novice users. These conventions, termed "secondary notations" in [113], are shown in [110] to correspond directly with the graphical pragmatics of [94].

More recent studies of the users of various other visual languages, notably visual programming languages, have highlighted similar usage of graphical pragmatics [112]. A major conclusion of this collection of studies is that the correct

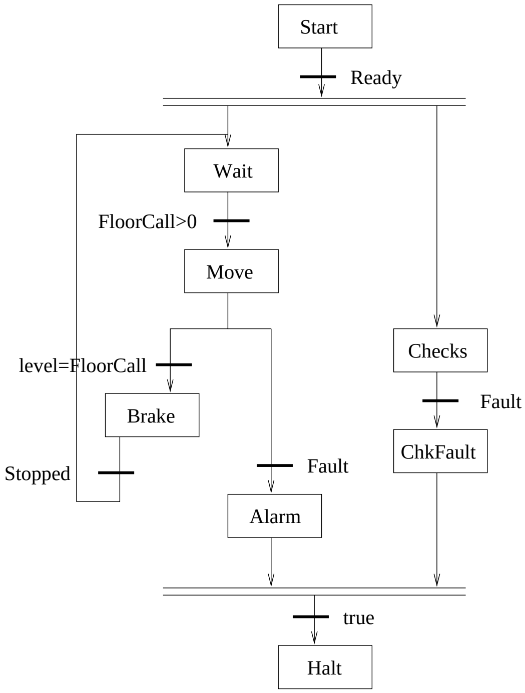


Figure 6.1: Example SFC diagram (lift controller).

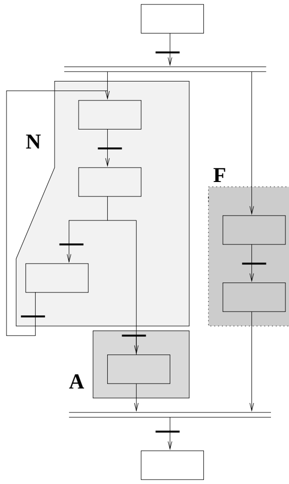
use of pragmatic features, such as layout in graph-based notations, is a significant contributory factor in the comprehensibility, and hence usability, of these representations.

6.2 Layout in SFC diagrams

A concrete application of our work is to the formalisation of diagrammatic (graph-based) languages for industrial embedded control software.

The SFC of Figure 6.1 is a simplified lift controller [1], adapted from a teaching example of "good" SFC design from [86]. While Figure 6.1 is a simplified version of the SFC from [86], nevertheless we have retained the layout of that original SFC, and note that it carries important information concerning the domain being represented. The main body of the SFC of Figure 6.1 is conceptually partitioned into the three regions illustrated in the following outline:

[1] "elevator" controller, for American readers



Region N is concerned with normal operation, A is an alarm-raising component and F performs fault detection (e.g. action "Checks" monitors the state of the lift and raises the boolean signal "Fault" whenever a fault occurs).

The following section concerns a general method for the specification of graph-based diagrams, which is illustrated in Section 6.4 through the formalisation of SFC diagrams. Thereafter we present how this formalisation both captures the layout features of SFCs such as that of Figure 6.1, and supports direct reasoning over the structure of such diagrams.

6.3 Specifying structure and layout

Recall from the previous chapter that a *graph* is a purely formal (i.e. mathematical) entity: a collection of "objects" and directed "arrows".

Let us call a class of diagrams *graph-based* if every diagram d in the class may be adequately abstracted (up to graph isomorphism) as a, possibly labelled, graph $G(d)$.

Example 6.1. SFC diagrams are graph-based. The objects of the corresponding graphs may be labelled with step names, transition conditions and the special symbols "F" (sequence divergence, or "fork"), "J" (sequence convergence, or "join"), "D" (divergence of concurrent sequences), "C" (convergence of concurrent sequences) standing for the four branching elements. Using these conventions, the graph corresponding to the lift controller of Figure 6.1 is illustrated in Figure 6.2. ■

Conversely, a graph G may be considered as a specification for (a class of) graph-based diagrams which, in particular, imposes no constraints on layout. Our aim here is to introduce an alternative formal specification of graph-based

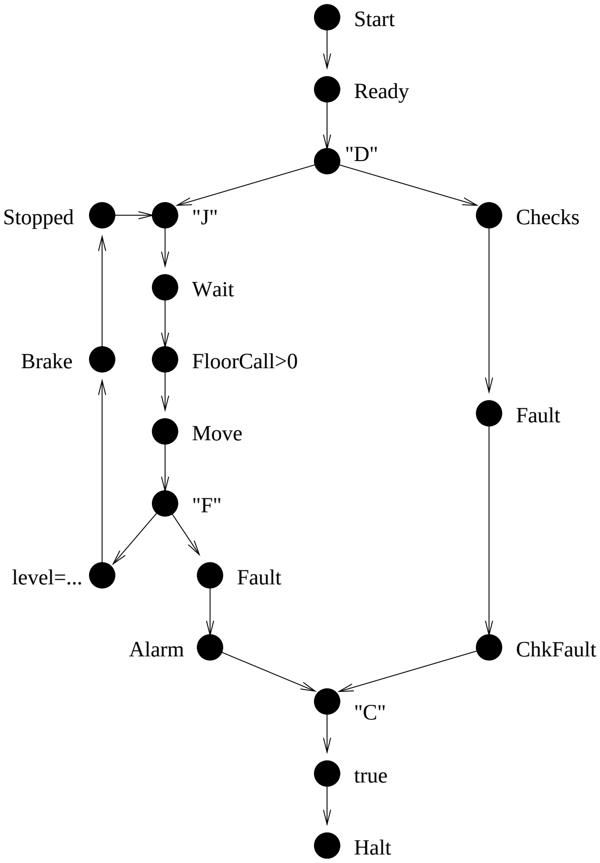


Figure 6.2: Graph for the SFC diagram of the lift controller.

diagrams which includes both structural and layout information. In general, these specifications will be expressions denoting arrows in a (suitably chosen) tensor category.

6.3.1 Diagrams of arrows in tensor categories

A remarkable recent development has been the introduction in [74] (and also [75]) of an intuitive diagrammatic representation for expressions denoting arrows in a variety of tensor categories. The resulting diagrams are modelled in terms of topological graphs, that is collections of nodes and edges. (In topological graphs, some edges may be only semi-attached or even completely detached from the nodes.) The nodes in these graphs are labelled with arrows in the category whereas the edges are labelled with objects. Diagrams are constructed inductively (starting from atomic ones corresponding to single arrows and using the composition ";" and tensor "⊗" of the category) and their layout is specified precisely by embedding the associated topological graphs into $\mathbb{R}^2$ or $\mathbb{R}^3$. For full details, the reader is referred to [74].

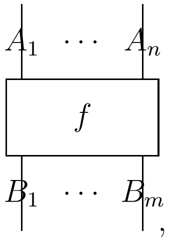
6.3.1.1 Basic, acyclic diagrams

To illustrate the representation, consider a tensor category $(\mathcal{C}, \otimes, e)$ and let

$$\begin{array}{rcl} A & = & A_1 \otimes \ldots \otimes A_n \\ B & = & B_1 \otimes \ldots \otimes B_m \\ C & = & C_1 \otimes \ldots \otimes C_k \\ D & = & D_1 \otimes \ldots \otimes D_j \end{array}$$

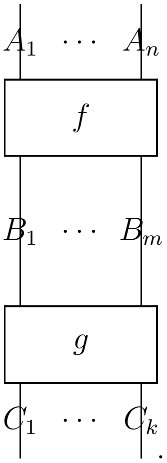
be objects in $\mathcal{C}$.

The diagram corresponding to an arrow $f : A \to B$ in $\mathcal{C}$ is:

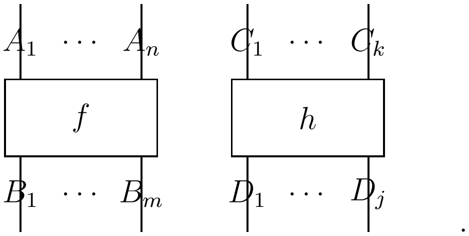


consisting of a single node and semi-attached edges as shown. Here we draw nodes generically as boxes (rectangles) but other choices of graphical symbols (e.g. circles, line segments etc.) are possible.

The diagram of $f; g$, the composite arrow of $f : A \to B$ and $g : B \to C$, results by joining correspondingly labelled edges as shown:



Finally, the diagram corresponding to $f \otimes h$, for f as before and $h : C \to D$, results in the juxtaposition of the component diagrams:



Consider now the axiom

$$(f \,;\, g) \otimes (f' \,;\, g') = (f \otimes f') \,;\, (g \otimes g')$$

of a tensor category and observe that both sides denote identical diagrams. Thus, the axiom is diagrammatically interpreted as a requirement that composition (;) and tensor ($\otimes$) should extend diagrams in completely orthogonal spatial dimensions. Together with the non-commutativity of both operations, which enforces a choice of direction for each, this is the key property allowing arrow expressions to unambiguously specify the layout of diagrams.

Special diagrams are defined for the distinguished arrows (e.g. identities, symmetries etc.) in the category: the diagrams of an identity arrow $1_A : A \to A$ and of a symmetry $\mathbf{c}_{A,B} : A \otimes B \to B \otimes A$ are given in Figure 6.3.

So far we have described the class of diagrams generated by expressions denoting arrows in an arbitrary symmetric tensor category. All diagrams in this class are *acyclic* in the sense that contain no edges linking a node to a preceding one (that is, one higher up in the diagram). For this reason, such diagrams are termed *progressive* in [74]; they can be constructed from top to bottom by progressively parsing the associated expression.

From now on, we shall often omit the labels on the edges of diagrams.

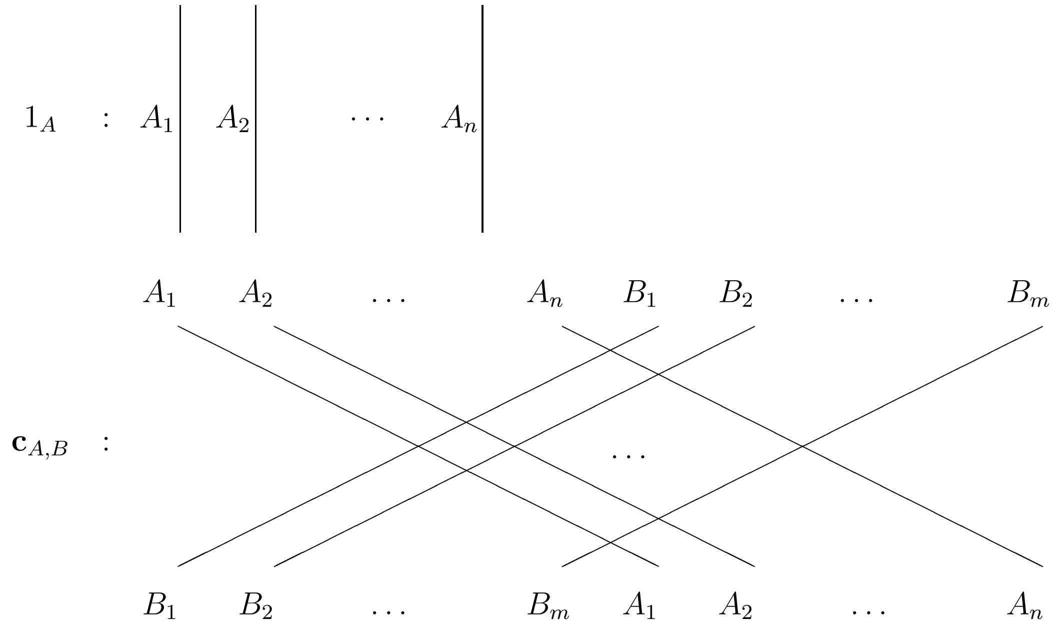

1_A :
A_1
A_2
$\cdots$
A_n
A_1 A_2 $\ldots$ A_n B_1 B_2 $\ldots$ B_m
$\mathbf{c}_{A,B}$:
$\ldots$
B_1 B_2 $\ldots$ B_m A_1 A_2 $\ldots$ A_n

Figure 6.3: Identity and symmetry diagrams ($A = A_1 \otimes \ldots \otimes A_n$ and $B = B_1 \otimes \ldots \otimes B_m$).

6.3.1.2 Transformations of layout

Beyond providing a precise formalisation of layout, the topological account of diagrams also captures the notion of *transforming* the layout of diagrams. For example, the diagrams of Figure 6.4 are all transformations of one another. Topologically, such transformations (or "deformations" in the terminology of [74]) are homeomorphic maps[2] (from the topological graph underlying the diagram to the usual topology on an appropriate subset of $\mathbb{R}^2$ or $\mathbb{R}^3$).

Transformations induce an equivalence relation on diagrams: two diagrams are regarded as being equivalent if they can be transformed into one another. A most remarkable result in [74] is that any two acyclic diagrams are equivalent in this sense *if and only if* their associated expressions can be proved equal using the axioms of a symmetric tensor category. Thus, the axioms of the category capture precisely the notion of layout transformation associated with this particular class of diagrams.

Example 6.2. From left to right, the diagrams in Figure 6.4 correspond to the expressions:

1. $f\,;(1 \otimes h)\,;(g \otimes 1)\,;k$
2. $f\,;(g \otimes h)\,;k$ and

[2] A homeomorphic map [134, 31] is an *isomorphism* of topological spaces.

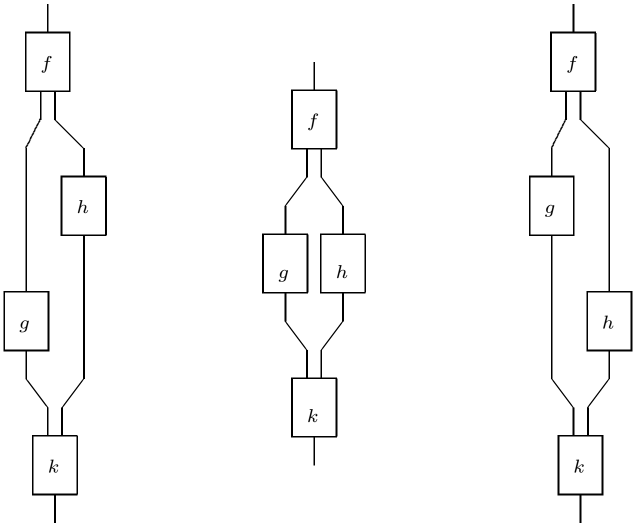


Figure 6.4: Three diagrams equivalent under transformation.

3. $f\,;\,(g \otimes 1)\,;\,(1 \otimes h)\,;\,k$.

By applying

$$(1 \otimes g)\,;\,(h \otimes 1) = (1; h) \otimes (g; 1)$$

(axiom 6 of Definition 5.8) and

$$g\,;\,1 = g \quad \text{and} \quad 1\,;\,h = h$$

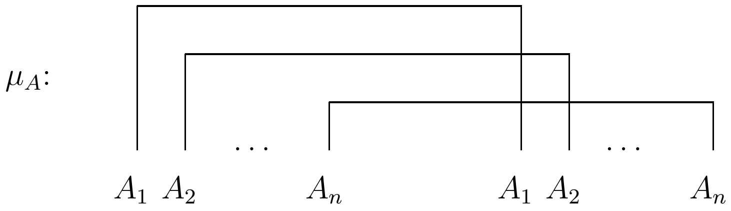
(axiom 3 of Definition 5.2) in that order one may now equate (1) to (2) as follows:

$$f\,;\,(1 \otimes h)\,;\,(g \otimes 1)\,;\,k = f\,;\,((1; g) \otimes (h; 1))\,;\,k = f\,;\,(g \otimes h)\,;\,k \ .$$

Similarly, any pair of expressions corresponding to two (distinct) diagrams in Figure 6.4 may be proved equal by using the axioms of a tensor category. ■

6.3.1.3 Cyclic diagrams

If, in addition, the category has units $\mu_A : e \to A \otimes A$ and counits $\eta_A : A \otimes A \to e$ (Definition 5.11), these are pictured as:



and

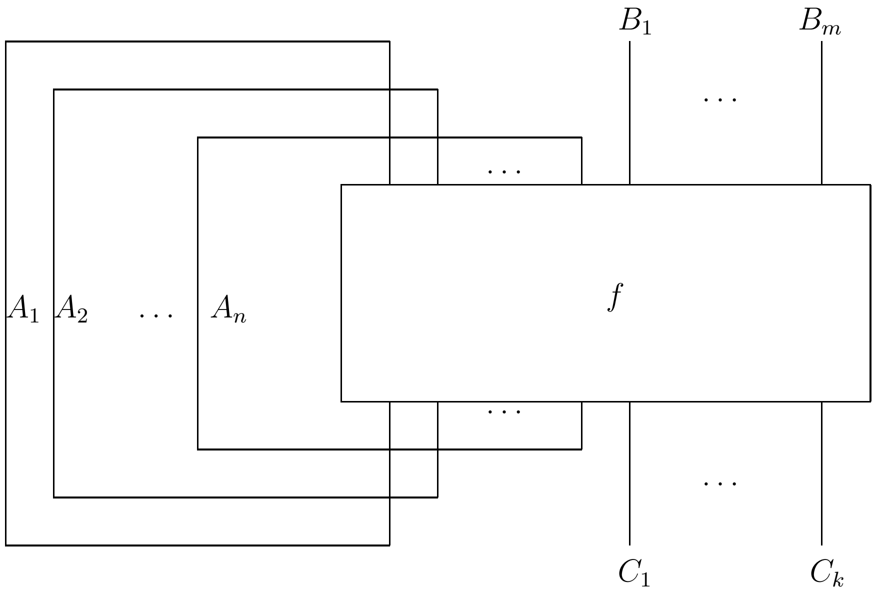
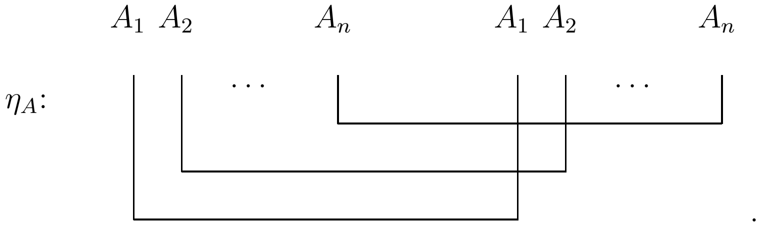


Figure 6.5: Diagram of $\mathrm{Tr}^A_{B,C}(f)$.



(Thus each of μ_A, η_A is represented as a bunch of n edges which are "bended" as shown but not permuted.)

Recalling (Proposition 5.3) that the canonical trace of $f : A \otimes B \to A \otimes C$ in such a category is obtained as

$$\mathrm{Tr}^A_{B,C}(f) \overset{\mathrm{def}}{=} (\mu_A \otimes 1_B)\,;\,(1_A \otimes f)\,;\,(\eta_A \otimes 1_C)\ \ ,$$

the diagram of $\mathrm{Tr}^A_{B,C}(f)$ is simply as given in Figure 6.5. Thus, a trace on a tensor category provides a canonical way of adding cycles (or "loops") to the diagrams representing arrows in the category. The extra axioms required to capture equivalence under transformation in this new class of diagrams are precisely the trace axioms of Definition 5.10. The diagrammatic representation of these axioms is illustrated in Figures 6.6 and 6.7.

6.4 Specifying SFC diagrams

Our proposal consists in using the arrows of an appropriately constructed tensor category to specify the structure and layout of diagrams in a variety of domain-

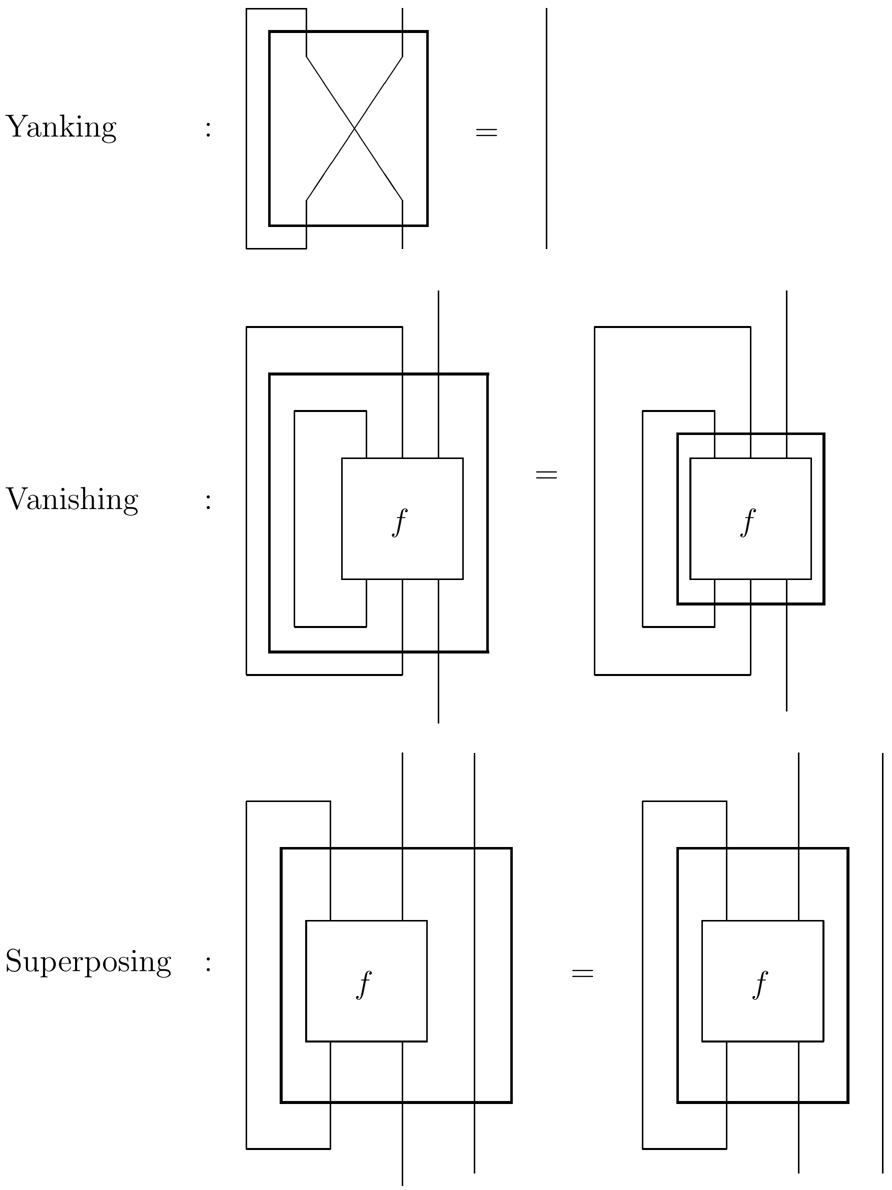


Figure 6.6: Diagrammatic presentation of the Yanking, Vanishing and Superposing axioms. (Extraneous thick-lined boxes have been added for clarity in indicating the sub-diagrams over which the trace operation is applied.)

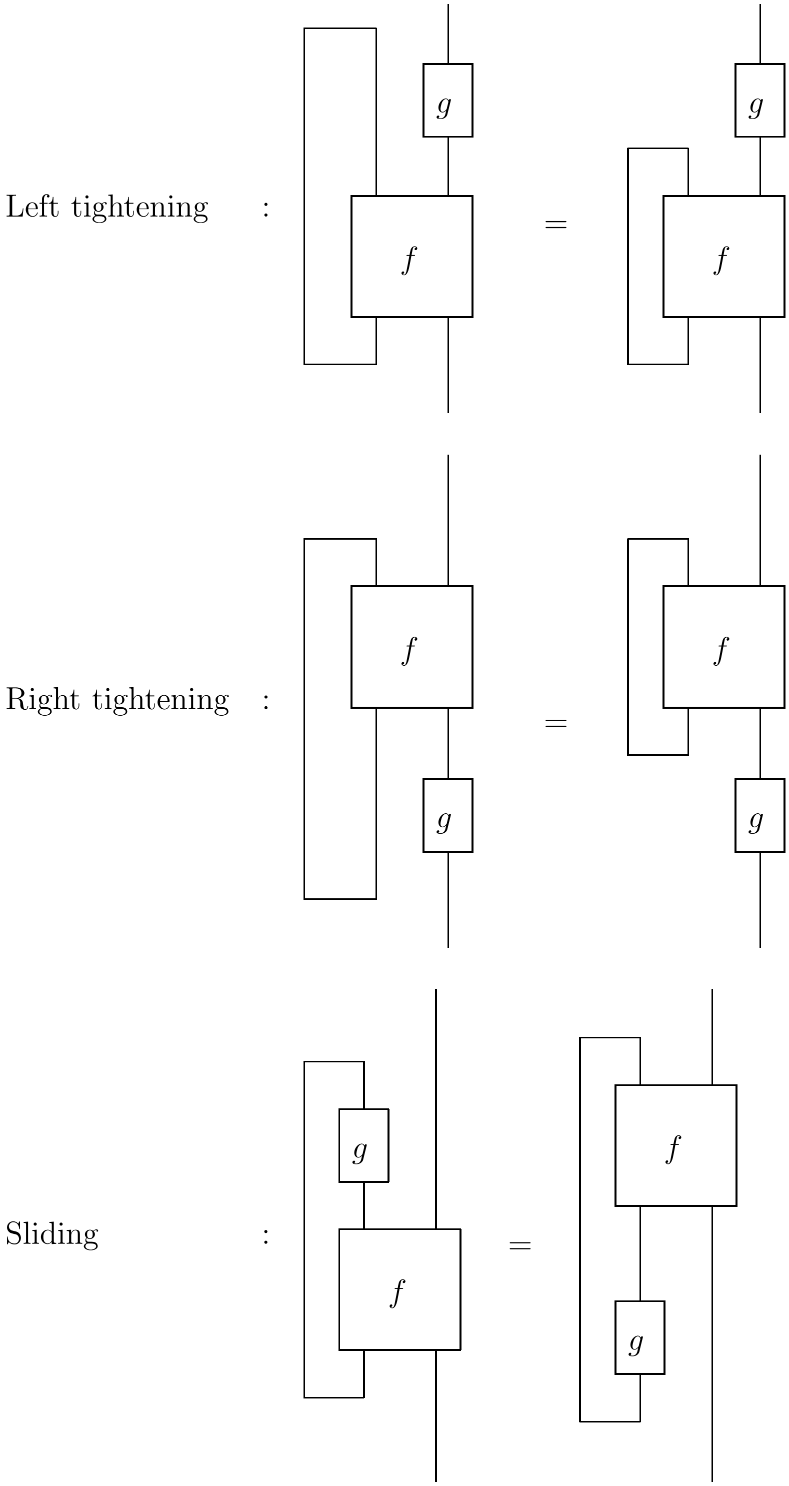


Figure 6.7: Diagrammatic presentation of the Tightening and Sliding axioms.

specific languages. The remainder of this chapter is devoted to illustrating the practical application of this proposal to the specification of SFC diagrams and to discussing the benefits that such an approach yields.

In the case of SFC diagrams, the arrows of the relevant category will be obtained from formal constructions which we call *metagraphs*:

Definition 6.1. In the context of the present chapter, a metagraph M is a particular labelling of a (finite) graph S. This labelling can be partial on arrows (i.e. some arrows of S may be left unlabelled). The graph S is called the shape of metagraph M. ■

A formal account of metagraphs (in terms of graph homomorphisms) will be provided in the following chapter.

Example 6.3. The graph

$$\begin{tikzcd} S:\ 1 \arrow[r, "f"] & 2 & 3 \arrow[l, "g"'] \end{tikzcd}$$

may be labelled by mapping: objects $1, 2$ to $\diamondsuit$; 3 to $\heartsuit$; and arrow g to $\spadesuit$. The result is the metagraph

$$\begin{tikzcd} \diamondsuit \arrow[r] & \diamondsuit & \heartsuit \arrow[l, "\spadesuit"'] \end{tikzcd}$$

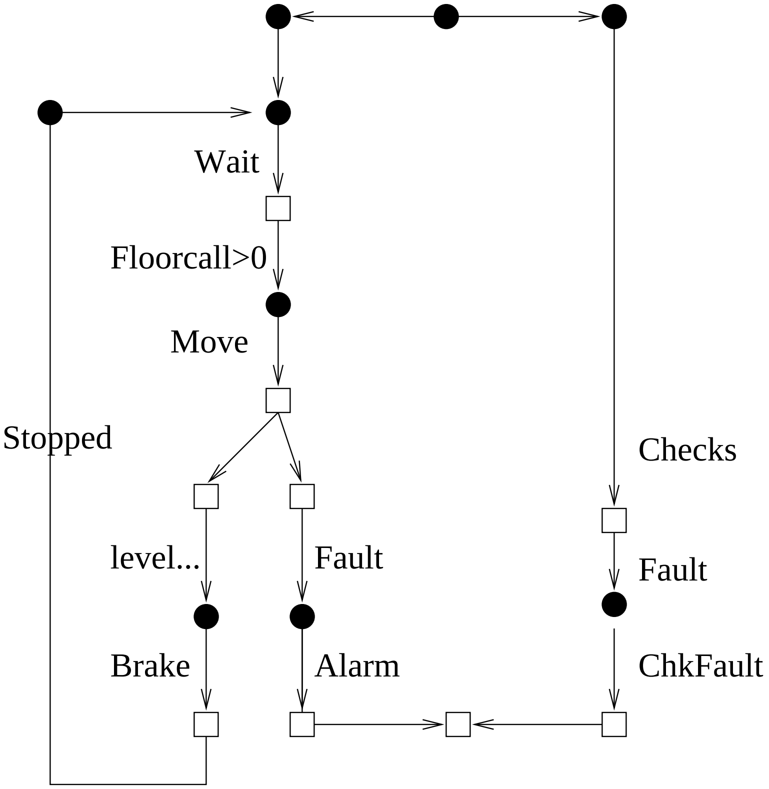
having the same "shape" as the original. ■

Let now $\mathbf{S}$ and $\mathbf{T}$ be the sets of all step names and transition conditions allowed in SFC diagrams. One way of specifying the structure of an SFC diagram is to construct a metagraph in which:

1. arrows $\bullet \xrightarrow{s} \Box$ labelled with elements of $\mathbf{S}$ correspond to steps;
2. arrows $\Box \xrightarrow{t} \bullet$ labelled with elements of $\mathbf{T}$ correspond to transitions;
3. (combinations of) unlabelled arrows $\Box \longrightarrow \Box$ and $\bullet \longrightarrow \bullet$ correspond to the branching constructs in SFCs.

Thus, the objects of the graph are labelled with either $\bullet$ or $\Box$. As we shall see, this labelling of objects defines a form of typing which, among other constraints, encodes the strict alternation of steps and transitions in SFC diagrams.

Example 6.4. One metagraph derived from the main body of the lift controller is:



in which $\bullet \longleftarrow \bullet \longrightarrow \bullet$ and $\Box \longrightarrow \Box \longleftarrow \Box$ correspond to the double horizontal lines in Figure 6.1 (i.e. the introduction and conclusion markers for concurrent sections in a diagram). ■

6.4.1 The tensor category of metagraphs

Having illustrated the correspondence of metagraphs to SFC diagrams, we proceed to outline how a tensor category can be constructed from metagraphs. The presentation appeals mostly to intuition and pictorial illustrations, deferring complete formalisations until the following chapter.

6.4.1.1 Concatenable metagraphs

Let M be a metagraph of shape S. Let the sources and sinks of M be respectively the sources and sinks (Section 5.1) of the shape graph S. One way of composing metagraphs unambiguously is to impose orderings on both their sinks and sources. These orderings are then regarded as defining "boundaries" through which metagraphs may be glued to each other.

Definition 6.2. In the context of the present chapter, a *concatenable metagraph* is a metagraph M together with an ordering of its sources and an ordering of its sinks. We shall let C range over concatenable metagraphs. ■

In the pictorial presentation of concatenable metagraphs we use numeric superscripts to indicate orderings on sources and subscripts for orderings on sinks:

Example 6.5. A concatenable version of the metagraph in Example 6.3 is:

$$\begin{tikzcd}
{}^1\diamondsuit \arrow[r] & \diamondsuit_1 & {}^2\heartsuit \arrow[l, "\spadesuit"']
\end{tikzcd}$$

■

6.4.1.2 Sequential composition

The sequential composition of C_1 and C_2 is only defined when there as many sinks in C_1 as there are sources in C_2 and, moreover, the chosen orders are such that the i-th sink of C_1 shares the same label with the i-th source of C_2. The resulting construction, denoted $C_1 \,;\, C_2$, identifies matching pairs of sources and sinks.

Example 6.6. Consider the concatenable metagraphs

$$C_1 = \begin{tikzcd}
{}^1\spadesuit \arrow[r, "f"] \arrow[dr, "j"] & \diamondsuit_1 \\
{}^2\diamondsuit \arrow[r, "g"] \arrow[dr, "k"'] & \spadesuit_2 \\
 & \diamondsuit_3
\end{tikzcd} \quad \text{and} \quad C_2 = \begin{tikzcd}
{}^1\diamondsuit \arrow[r, "g"] & \spadesuit_1 \\
{}^2\spadesuit \arrow[r, "f"] & \diamondsuit_2 \\
{}^3\diamondsuit \arrow[ur, "k"'] &
\end{tikzcd}$$

$C_1 \,;\, C_2$ is the concatenable metagraph:

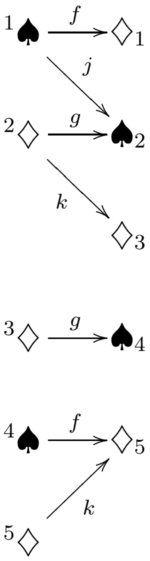
$$\begin{tikzcd}
{}^1\spadesuit \arrow[r, "f"] \arrow[dr, "j"] & \diamondsuit \arrow[r, "g"] & \spadesuit_1 \\
{}^2\diamondsuit \arrow[r, "g"] \arrow[dr, "k"'] & \spadesuit \arrow[r, "f"] & \diamondsuit_2 \\
 & \diamondsuit \arrow[ur, "k"'] &
\end{tikzcd}$$

■

6.4.1.3 Parallel composition

The parallel composition of C_1 and C_2, denoted $C_1 \otimes C_2$, results in the "disjoint union" of the two concatenable metagraphs, pictured as juxtaposition. Moreover the relative orderings on the sinks and sources in each of the component metagraphs are preserved.

Example 6.7. For C_1 and C_2 as in Example 6.6, $C_1 \otimes C_2$ is simply:

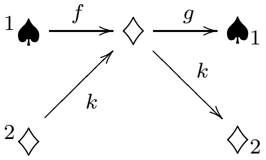


■

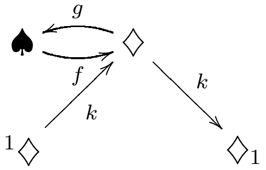
6.4.1.4 Loops

Let C be a concatenable metagraph in which the first sink shares the same label as the first source. Then, one can identify these two "matching" endpoints to create a loop over C. The resulting metagraph is denoted $\circlearrowright C$.

Example 6.8. Consider the concatenable metagraph C,



in which the first source matches the first sink. The concatenable metagraph $\circlearrowright C$ is:



■

6.4.1.5 Types for concatenable metagraphs

The orderings of the sinks and sources in a concatenable metagraph C may be succinctly presented as strings (i.e. ordered sequences) of labels. So, from the sources of C one obtains a string, say α, in which the i-th element is the label on the i-th source. A string, say β, is similarly obtained from the sinks. Thus, C may be seen as being "of type" $\langle \alpha, \beta \rangle$, and one writes $C : \alpha \rightarrow \beta$.

Example 6.9. In Example 6.6, $C_1 : \spadesuit\diamondsuit \rightarrow \diamondsuit\spadesuit\diamondsuit$ and $C_2 : \diamondsuit\spadesuit\diamondsuit \rightarrow \spadesuit\diamondsuit$.

Now, one may reformulate the condition in the definition of sequential composition as: $C_1; C_2$ is only defined when $C_1 : \alpha \rightarrow \beta$ and $C_2 : \beta \rightarrow \gamma$. Then, $C_1; C_2 : \alpha \rightarrow \gamma$. Types are therefore a useful formalisation of constraints subject to which component metagraphs may be put together to form composites.

Given a string α of $n \geq 0$ labels, one can construct a concatenable metagraph denoted 1_α by labelling the graph consisting of objects $\{1, \ldots, n\}$ and no arrows so that object i is mapped to the i-th element of α. For instance, $1_{\spadesuit\diamondsuit\spadesuit}$ is the concatenable metagraph:

$$\begin{array}{c} {}^{1}\spadesuit_{1} \\ {}^{2}\diamondsuit_{2} \\ {}^{3}\spadesuit_{3} \end{array} \quad .$$

Clearly, the type of each 1_α is $\alpha \rightarrow \alpha$.

6.4.1.6 Equational laws

Relying on intuition alone, one quickly establishes the following list of plausible identities among concatenable metagraphs, holding whenever all composite expressions involved are well defined (i.e. well-typed):

1. $1_\alpha; C = C$ and $C; 1_\beta = C$, for $C : \alpha \rightarrow \beta$

2. $C_1; (C_2; C_3) = (C_1; C_2); C_3$

3. $1_\alpha \otimes 1_\beta = 1_{\alpha\beta}$ ($\alpha\beta$ denotes the concatenation of α and β)

4. $C_1 \otimes (C_2 \otimes C_3) = (C_1 \otimes C_2) \otimes C_3$

5. $(C_1 \otimes C_2); (C_1' \otimes C_2') = (C_1; C_1') \otimes (C_2; C_2')$

6. $\circlearrowright (C \otimes 1_\alpha) = (\circlearrowright C) \otimes 1_\alpha$

7. $\circlearrowright ((1_l \otimes C'); C) = C'; (\circlearrowright C)$, where l is a single label

8. $\circlearrowright (C; (1_l \otimes C')) = (\circlearrowright C)\, ; C'$, where l is a single label

9. $\circlearrowright (C\, ; (C' \otimes 1_\alpha)) = \circlearrowright ((C' \otimes 1_\alpha)\, ; C)$.

The first two laws establish a category having strings of labels as objects and as arrows from α to β all concatenable metagraphs of type $\alpha \to \beta$. Laws 3, 4 and 5 are a subset of the conditions required to make this a tensor category. Finally, the reader is invited to examine the similarity of the remaining laws to the Superposing, Left and Right tightening and Sliding axioms for traces (Definition 5.10).

In the following chapter we shall extend this list of equations to obtain a (symmetric) tensor category of concatenable metagraphs equipped with a trace generalising $\circlearrowleft$. This will establish fully the connection between expressions involving concatenable metagraphs and diagrams. For the time being, we take the result as granted and concentrate on application.

6.4.2 SFC expressions

Every formalised SFC diagram may now be expressed syntactically in terms of a few basic metagraphs and the operators $\circlearrowleft$, $\otimes$ and ;.

Let s denote the concatenable metagraph ${}^1\bullet \xrightarrow{s} \Box_1$, $s \in \mathbf{S}$, and t denote ${}^1\Box \xrightarrow{t} \bullet_1$, $t \in \mathbf{T}$. In particular, the use of labels $\bullet$ and $\Box$ acts as a typing scheme encoding the strict alternation of steps and transitions: expressions such as $s; s'$ or $t; t'$ ($s, s' \in \mathbf{S}$, $t, t' \in \mathbf{T}$) are forbidden as they have no meaning as SFC diagrams.

We also introduce concatenable metagraphs

$$\Delta_\bullet \overset{\text{def}}{=} \bullet_1 \longleftarrow {}^1\bullet \longrightarrow \bullet_2 \quad \text{and} \quad \nabla_\bullet \overset{\text{def}}{=} {}^1\bullet \longrightarrow \bullet_1 \longleftarrow {}^2\bullet$$

and corresponding versions ($\Delta_\Box$, $\nabla_\Box$) for $\Box$. In particular $\Delta_\bullet$ and $\nabla_\Box$ stand for the introduction and conclusion markers in concurrent sections, whereas $\nabla_\bullet$ and $\Delta_\Box$ will stand for "joins" and "forks" of links. We also abbreviate $1_\bullet$ ($= {}^1\bullet_1$) as just 1.

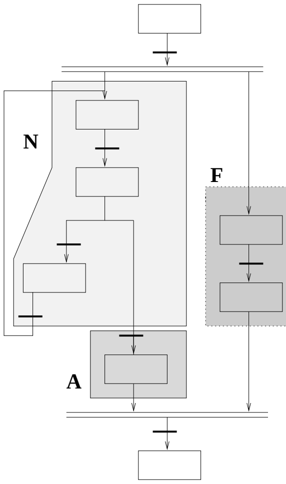
The metagraph of Example 6.4 may now be expressed as

$$\mathit{LiftBody} \equiv \Delta_\bullet; [(\circlearrowleft N); A \otimes F]; \nabla_\Box \ ,$$

where:

$$\begin{aligned} N &\equiv \nabla_\bullet; \mathrm{Wait}; \mathrm{FloorCall} > 0; \mathrm{Move}; \Delta_\Box; \\ &\quad\ (\mathrm{level}\ldots; \mathrm{Brake}; \mathrm{Stopped} \otimes 1) \\ A &\equiv \mathrm{Fault}; \mathrm{Alarm} \\ F &\equiv \mathrm{Checks}; \mathrm{Fault}; \mathrm{ChkFault}\ . \end{aligned}$$

The expressions N, A and F correspond to the regions identified at the end of Section 6.2:



Finally, the expression for the entire controller is:

$$\mathit{Lift} \equiv \mathrm{Start};\mathrm{Ready};\mathit{LiftBody};\mathrm{true};\mathrm{Halt} \ \ .$$

Any given, non-atomic diagram can be constructed or decomposed in a variety of ways. For instance, an alternative way of decomposing the main body of our lift controller is provided by the expression

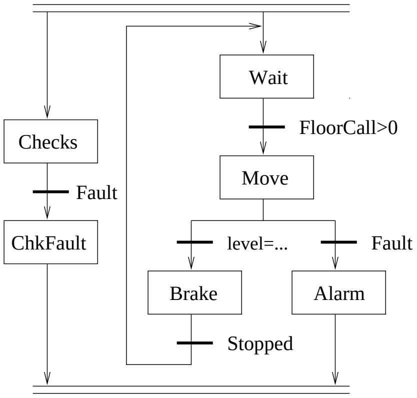
$$\Delta_{\bullet};[((\circlearrowright N)\otimes(\mathrm{Checks};\mathrm{Fault}));(A\otimes\mathrm{ChkFault})];\nabla_{\Box} \ \ .$$

This view of the diagram, which might be appropriate to a particular kind of analysis, may now be proved equivalent to the one given by expression *LiftBody* above by applying the equational law

$$(C_1\otimes C_2);(C_1'\otimes C_2')=(C_1;C_1')\otimes(C_2;C_2') \ \ .$$

Thus, the utility of the equational laws lies in identifying equivalent views of the same representation, such as those resulting from transformations of layout or from different perceptions of structure.

Consider now an alternative SFC diagram for the main body of our controller:



The layout of this diagram contains less pragmatic information regarding the distinction between which parts of the system are concerned with normal and exceptional operation. In the original diagram of Figure 6.1, both regions A and F concerning exceptional operation are distinguished from the region N. The alternative diagram, however, fails to make this distinction, suggesting a partitioning in only two regions, the one on the right responsible for both normal and alarm-raising behaviour.

Formalised as graphs, both diagrams would be indistinguishable. (Both diagrams comprise the same steps and transitions, which are in both cases interconnected in the same way.) In our specification language, however, the new version of the controller's body diagram would be obtained from the expression:

$$\mathit{LiftBody}' \equiv \Delta; [F \otimes (\circlearrowright X)]; \nabla \ \ ,$$

where:

$$\begin{aligned} X \ \ \equiv \ \ & \nabla; \mathrm{Wait}; \mathrm{FloorCall} > 0; \mathrm{Move}; \Delta; \\ & [(\mathrm{level}\ldots; \mathrm{Brake}) \otimes A]\,; (\mathrm{Stopped} \otimes 1) \ \ , \end{aligned}$$

and A, F are as before. The metagraph denoted by this expression is the same as the one in Example 6.4.

Remark. *Readers familiar with an earlier version of this chapter [48] will notice some minor differences resulting from the use here of "1" in place of* $\mathbf{1} \overset{\mathrm{def}}{=} {}^{1}\bullet \longrightarrow \bullet_{1}$.

6.5 Representations and tasks

A significant determiner of what makes a particular representation *effective* is that it should simplify various reasoning tasks. One benefit that certain diagrammatic representations offer to support this is the potential to directly capture pertinent aspects of the represented artifact (whether this be a concrete artifact or some abstract concept). To clarify this argument, which is explored in [50, 47], we must explain what is meant here by the terms "direct" and "pertinent".

Firstly, diagrammatic relations can often be directly semantically interpreted. This is to say that certain diagrammatic relations exhibit intrinsic properties, such as transitivity or symmetry, and these may be exploited in a systematic way by choosing to represent some aspect of the represented artifact with a diagrammatic relation which has matching intrinsic properties. For example, the operation of the lift controller in our example is conceptually partitioned into three modes: of normal, alarm-raising and fault checking behaviour. The layout of the SFC diagram of Figure 6.1 is such that, for any given step or transition, membership of one these modes is represented as membership of an identifiable region of the graph (one of the regions N, A or F above). Thus, a conceptual aspect of the artifact (membership of some behavioural mode) is directly captured by a representing relation in the diagram with matching logical properties (membership of a spatial region in the plane). Note that, in this case as with the expert designers of CAD diagrams and visual programs studied in [113, 112], it is the pragmatic features of the diagram (layout being chosen so as to suggest conceptual regions) which are exploited to carry this information. Indeed, in the original SFC diagram from [86], of which our example is a simplification, these regions were strikingly well delineated.

Secondly, by "pertinent" aspects of the represented artifact, we refer to those aspects which are relevant to particular reasoning tasks. We argue that reasoning is strongly influenced by the structure of the representation within which one reasons. Where the structure of a representation matches the primary concepts over which one must reason, reasoning is made easier. Conversely, having the "wrong" structure in a representation will interfere with reasoning, making it more difficult. This argument is supported by a number of empirical studies of users employing different representations for similar tasks; as in, for example, studies of various diagrammatic representations used in solving logical syllogisms [130], and of alternative representations employed in solving the Tower of Hanoi problem [148].

One of the major tasks that SFC notations intend to support is the inference (by system designers) of which sequences of states may the system exhibit. Desirable such sequences are formulated in terms of system properties, prominent among which are *safety* properties. For instance, appropriate for our lift controller is property *Safe* expressed as: "Assuming no faults, the lift always stops before the next call is attended."

Example 6.10. The diagram of Figure 6.1 exhibits property *Safe* because:

1. Once the main loop is entered, the assumption of fault-freeness implies that control is retained within the loop; and

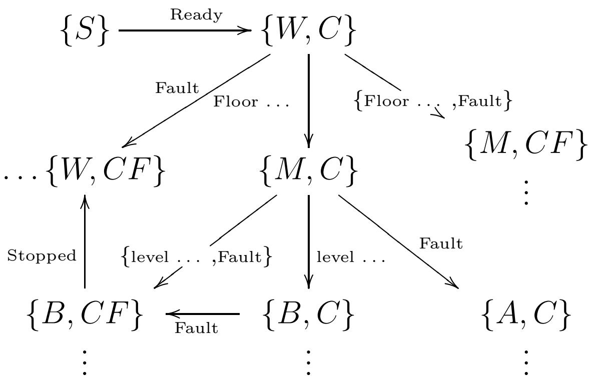
2. the loop forms a single path from step "Move" to itself that includes condition "Stopped". ■

Crucial to part (2) of this argument is the observation that paths in the diagram correspond semantically to temporal orderings of events: if only a single path exists from any current step A to step B and condition t appears along the path, the next activation of B must be preceded by an occurrence of t.

A desirable software engineering goal is to support the formalisation of informal arguments, such as the above. We argue that for a formalisation to be effective, as with an effective diagrammatic representation, it should accurately structure those aspects of the represented artifact which are pertinent to the required reasoning tasks. For example, essential to the informal argument in the preceding example is the ability to focus precisely on the part of the diagram which is responsible for property *Safe*. That is, focusing on the loop and excluding the alarm-raising and fault-checking parts of the diagram. A formalisation which does not readily permit a similar structuring, will clearly be less effective than one which does.

Consider that there are numerous, less direct ways of capturing the semantics of SFCs; the most common being to enumerate all possible states of the SFC in a transition system [131].

Example 6.11. A transition system modelling the behaviour of the lift controller has as states all reachable sets of actions. Its transitions are possible combinations (i.e. sets) of conditions. A small part of this transition system is given below (with step names truncated to their initials):



This resulting transition system is typical of a formalisation which obscures the structure necessary to the informal argument of Example 6.10. This is because paths in the transition system result from the interweaving of events belonging to several concurrent components in the SFC. In contrast to the informal argument of Example 6.10, when arguing properties such as *Safe* on the model of Example 6.11 it is generally hard to exclude behaviour originating in parts of the system which are otherwise unrelated to the property in question. While it could be argued that this is hardly a problem for small SFCs, the size of a transition system grows exponentially to the number of concurrent components in the SFC.

6.6 Reasoning on diagrams

Our algebra enables the formalisation of reasoning arguments which are directed by the structure of the diagrammatic representation of the system(s), as we illustrate next.

Traditional approaches to formal reasoning favour behavioural models and are typically less concerned with user-oriented representations. Thus, given a system expressed as diagram d and a property p, traditional approaches typically consist of:

1. obtaining some behavioural model $\mathcal{M}(d)$ of d, such as a function, a transition system, etc.;
2. formalising p as a formula $\phi(p)$ in some suitable logic; and
3. verifying whether $\mathcal{M}(d) \models \phi(p)$, i.e. whether the model satisfies the formula.

Example 6.12. Property *Safe* may be formalised as a temporal formula [131]. What is important here is the overall structure of the formula, $\alpha \implies \sigma$, where α expresses an assumption about the computation (here "always not Fault") and σ expresses a commitment of the system. For the purposes of illustration, consider the (rather crude) example:

$$\begin{aligned}\phi(\mathit{Safe}) \overset{\text{def}}{=} \mathcal{A}(\neg\text{Fault}) \implies (\mathcal{I}(\text{Stopped}) \implies \\ \mathcal{A}(\text{Stopped}\ \mathcal{BN}\ (\text{FloorCall} > 0\ \wedge\ \text{Move})))\end{aligned} \tag{6.1}$$

where $\mathcal{A}(\psi)$ is interpreted as "always ψ", $\mathcal{I}(\phi)$ as "initially ϕ", $\psi_1 \mathcal{BN} \psi_2$ as "ψ_1 before next time ψ_2 holds" and $\implies$ as "implies". ■

Roughly speaking, our approach attempts to substitute structural (i.e. algebraic) models of diagrams for behavioural models in step (1) above. For example, if $C(d)$ is a concatenable metagraph expression corresponding to an SFC diagram d, and formula $\phi(p)$ expresses a property, the reasoning problem (step (3) above) is reformulated as $C(d) \models \phi(p)$. In terms of our example property *Safe* and the main body of our lift controller:

$$\mathit{LiftBody} \models \phi(\mathit{Safe}) \ . \tag{6.2}$$

The implicit inferential "short-cuts" in the informal argument of Example 6.10 may now be formalised, and thus justified, by means of inference rules. For instance, one rule views concurrency as the conjunction of the components' respective properties:

Rule 1: If $C \models \phi$ and $C' \models \phi'$, then $\Delta; (C \otimes C'); \nabla \models \phi \wedge \phi'$.

Another rule eliminates part of a diagram which is inaccessible under given assumptions:

Rule 2: If $[C \models \mathcal{A}(\neg\psi) \implies \phi]$ and $[t \implies \psi]$, then

$$C; (t; C') \models \mathcal{A}(\neg\psi) \implies \phi$$

Example 6.13. Starting with goal (6.2), equivalently written as

$$\Delta; [(\circlearrowright N); A \otimes F]; \nabla \models \phi(\mathit{Safe}) \wedge \mathit{true} \ \ ,$$

and applying Rule 1 yields sub-goals $(\circlearrowright N); A \models \phi(\mathit{Safe})$ and $F \models \mathit{true}$, the second of which is trivial. By Rule 2, we now discard the alarm-raising component to concentrate on the part precisely responsible for our property: $\circlearrowright N \models \phi(\text{safe})$. ■

Writing $\vdash C = C'$ whenever C and C' may be shown equal using the equational laws for concatenable metagraphs (Section 6.4.1), another interesting rule arises which permits the interchange of different structural or layout views of the same diagram within the course of a proof:

Rule 3: If $\vdash C = C'$ and $C' \models \phi$ then $C \models \phi$.

The soundness of rules such as those above must eventually be established wrt. some behavioural model. This obligation, however, lies with the developers of the visual language in question and not with the users.

6.7 Discussion

The main problem addressed in this chapter regards the formalisation of graph-based notations in a way which provides layout information. One way of specifying both the structure and layout of graph-based diagrams is to construct a tensor category from the basic elements of such diagrams. We have illustrated this by associating the basic elements of SFC diagrams (steps, transitions and branching elements) to such a category of metagraphs.

In pure algebra and theoretical computer science, this idea has been previously applied to the development of graphical notations for various calculi [37] and, more recently, to the graphical illustration of programming language semantics [73]. The novelty of our work lies in exploring the potential of this idea in the specification and modelling of (a large class of) diagrammatic (domain-specific) languages and, particularly, in its application to a concrete notation from software engineering (SFC diagrams).

The main difficulty in developing a category suitable for SFC diagrams was the invention of a typing scheme which, despite its remarkable simplicity, was not immediately obvious. This was partly due to the existence of various alternative schemes (cf. the discussion section in the following chapter). By contrast, the development of a graphical representation for a calculus builds upon an *á priori* established type discipline (expressed in terms of sorts in an algebraic signature or objects in a category, depending on the formulation of the calculus). Many diagrammatic programming notations are also readily typed: the type of a function block diagram, for instance, is a pair. The components of the pair are given by the lists of the block's input and output ports. A tensor category appropriate for function block diagrams is more-or-less apparent. The following chapter elaborates on how metagraphs emerged in connection with SFC diagrams.

Formalising the layout of an arbitrarily drawn (graph-based) diagram is a difficult task, one which requires at least some use of topology. From a practical standpoint, such a formalisation would be too complex and concrete to serve as an abstract syntax description. Instead, our approach addresses the problem in the opposite direction. One starts with a set of expressions denoting entities which, by virtue of satisfying a particular pattern of equational laws (those of a tensor category), map to stratified layouts of graph-based diagrams. The choice of the objects and arrows making up the category is merely a *parameter* to our approach, so long as they bear an understood relationship to the diagrams in question. Even for SFC diagrams, our choice of metagraphs is far from unique. It is conceivable that a different, perhaps simpler choice could have been made.

Our approach also suggests a particular interaction mode for visual program editors. We envisage an editing mode which, by contrast to “free-hand” drawing, produces highly stylised layouts of diagrams. In such a mode, diagrams are constructed by juxtaposing (in both dimensions) instances of pre-defined, parametrised “tiles”. In other words, the drawing space is “quantised” to form a grid. Whilst still allowing considerable freedom in choosing a suitable layout, this approach results in diagrams which are trivial to parse.

Graph-based notations form a significant subclass of visual programming languages and (software) engineering notations. Most typical accounts of such notations concentrate on concrete syntax (e.g. in terms of productions in graph grammars) and ignore the pragmatic aspects of graphs, most notably spatial layout. However, such pragmatic aspects – again, most notably layout – have been shown to be a significant contributing factor to the success and popularity of these notations.

This chapter has presented an algebraic, and therefore formal, account of graph-based notations which exhibits a considerable sensitivity to pragmatic features such as layout. Furthermore, our account enables formal reasoning over such representations in a manner which clearly parallels the informal, "intuitive" arguments which diagrams so often support.

Chapter 7

Metagraphs for SFC diagrams

In this chapter we provide insight into the choice of metagraphs as formal specification entities for SFC diagrams and rigorously demonstrate that concatenable metagraphs form a tensor category with a trace. In doing so, we point out which elements of our approach may be generalised to other situations.

7.1 Intuition behind metagraphs

7.1.1 SFC nets

One commonly employed abstraction of SFC programs (see, e.g. [54], page 69) is in terms of *elementary nets* [136]:

Definition 7.1. A (elementary) *net* is a triple (S, T, F) where

- S and T are sets such that $S \cap T = \emptyset$
- $F \subseteq (S \times T) \cup (T \times S)$ is a relation.

The elements of S and T are called *places* and *transitions* respectively. F is called the *flow relation*.

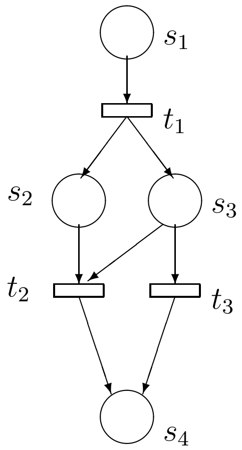
We shall make the (harmless) assumption that $*$ is a special element occurring neither as a place nor as a transition in any of the nets considered here. For any net (S, T, F) introduce the following convenient notation:

$$\begin{aligned} F^* &\overset{\text{def}}{=} F \cup \{\langle *, x\rangle \mid x \in S \cup T,\ \nexists y.\ \langle y, x\rangle \in F\} \cup \\ &\phantom{\overset{\text{def}}{=}} \{\langle x, *\rangle \mid x \in S \cup T,\ \nexists y.\ \langle x, y\rangle \in F\}\ , \\ pre(x) &\overset{\text{def}}{=} \{y \mid \langle y, x\rangle \in F^*\}, x \in S \cup T\ , \\ post(x) &\overset{\text{def}}{=} \{y \mid \langle x, y\rangle \in F^*\}, x \in S \cup T\ . \end{aligned}$$

Example 7.1. Consider the net with places $S = \{s_1, s_2, s_3, s_4\}$, transitions $T = \{t_1, t_2, t_3\}$ and flow relation:

$$F = \{\langle s_1, t_1\rangle, \langle t_1, s_2\rangle, \langle t_1, s_3\rangle, \langle s_2, t_2\rangle, \langle s_3, t_2\rangle, \langle s_3, t_3\rangle, \langle t_2, s_4\rangle, \langle t_3, s_4\rangle\} \ .$$

There is an informal, diagrammatic presentation of nets in which places are depicted as circles, transitions as boxes and pairs $\langle x, y\rangle$ as directed arcs from x to y. The diagram of the net (S, T, F) above is thus:



It is immediate from this diagram that $pre(s_1) = \{*\}$, $post(s_4) = \{*\}$, $pre(s_4) = \{t_2, t_3\}$ and $post(t_1) = \{s_2, s_3\}$. ■

To capture SFC programs, one needs to additionally specify labellings of the places and transitions in elementary nets with step names (or actions) and transition conditions respectively. (It seems, though, that this point is glossed over in other treatments of SFCs). Moreover there are many nets, including the one in the preceding example, which cannot arise from legal SFC programs. To overcome these deficiencies, let us introduce the following:

Definition 7.2. Let $\mathbf{S}$ be the set of step names and $\mathbf{T}$ the set of condition expressions allowed in SFC programs. (Assuming $\mathbf{S}$ and $\mathbf{T}$ disjoint.) An *SFC-net* $\nu = (S, T, F, l)$ consists of:

- a finite net (S, T, F)
- a labelling function $l : S \cup T \to \mathbf{S} \cup \mathbf{T}$

subject to the conditions:

1. $l(s) \in \mathbf{S}$ for all $s \in S$ and $l(t) \in \mathbf{T}$ for all $t \in T$
2. for all $x, y \in S \cup T$, $post(x) \cap post(y) \neq \emptyset$ implies $post(x) = post(y) (\neq \emptyset)$. ■

Key to the development of metagraphs as specifications of SFC diagrams has been the observation of an appealing correspondence between SFC-nets and graph homomorphisms into a graph $\mathcal{N}$ constructed from the labels. Specifically, $\mathcal{N}$ has objects $\{\bullet, \square\}$, an arrow $s : \bullet \to \square$ for each $s \in \mathbf{S}$ and an arrow $t : \square \to \bullet$ for each $t \in \mathbf{T}$. The correspondence may now be stated precisely as follows:

Proposition 7.1. *Let* $\mathbf{SNet}$ *be the set of SFC-nets and* $\mathbf{H}_{\mathcal{N}}$ *be the set of all graph homomorphisms* $h : G \to \mathcal{N}$*, where* G *varies. There exist functions* $H : \mathbf{SNet} \to \mathbf{H}_{\mathcal{N}}$ *and* $N : \mathbf{H}_{\mathcal{N}} \to \mathbf{SNet}$ *such that* $N(H(\nu)) = \nu$ *for all SFC-nets* ν*. That is,* N *is a one-sided inverse of* H*.*

Proof. (Sketch) Given an SFC-net $\nu = (S, T, F, l)$ define an equivalence relation $\sim$ on F^* by $\langle x_1, y_1 \rangle \sim \langle x_2, y_2 \rangle$ iff $\mathit{post}(x_1) = \mathit{post}(x_2)$. As usual, denote the set of all equivalence classes in F^* as $F^*/\!\sim$.

Thus, given any equivalence class $f \in F^*/\!\sim$ and any two elements $\langle x, y \rangle, \langle x', y' \rangle$ in f, one has $\mathit{post}(x) = \mathit{post}(x')$. Write $\mathit{post}(f)$ for the set $\mathit{post}(x)$, where $\langle x, y \rangle$ is any element of f. Using the definition of SFC-nets, it is easy to check that, given $f, f' \in F^*/\!\sim$, $\mathit{post}(f) \cap \mathit{post}(f') \neq \emptyset$ implies $f = f'$. Consequently, for any $y \in S \cup T$, there is *unique* $f \in F^*/\!\sim$ such that $y \in \mathit{post}(f)$.

Define now, for each SFC-net $\nu = (S, T, F, l)$, a graph $G(\nu)$ having

- $F^*/\!\sim$ as its set of objects,
- $S \cup T$ as its set of arrows,
- $\partial_0(x)$ given by the unique $f \in F^*/\!\sim$ such that $x \in \mathit{post}(f)$; and
- $\partial_1(x)$ given by the unique $f \in F^*/\sim$ such that $\langle x, y \rangle \in f$ for some y. (Uniqueness guaranteed by the definition of $F^*/\!\sim$.)

The idea is that $\partial_1(x) = \partial_0(y)$ in $G(\nu)$ whenever $\langle x, y \rangle \in F$.

The graph homomorphism $H(\nu)$ from $G(\nu)$ to $\mathcal{N}$ is now defined as:

- $H(\nu)(x) = l(x)$ for $x \in S \cup T$
- $H(\nu)(\partial_0(x)) = \bullet$ if $x \in S$ and $H(\nu)(\partial_0(x)) = \square$ if $x \in T$
- $H(\nu)(\partial_1(x)) = \square$ if $x \in S$ and $H(\nu)(\partial_1(x)) = \bullet$ if $x \in T$.

Let now $h : G \to \mathcal{N}$ be a graph homomorphism, where $G = (O, A, \partial_0, \partial_1)$. Define $N(h)$ to be (S', T', F', l') where:

- $S' = \{x \mid x \in A, h(x) \in \mathbf{S}\}$

- $T' = \{x \mid x \in A, h(x) \in \mathbf{T}\}$
- $F' = \{\langle x, y \rangle \mid x, y \in A, \partial_1(x) = \partial_0(y)\}$
- $l'(x) = h(x)$ for all $x \in A$.

$N(h)$ can be seen to satisfy the conditions in the definition of SFC-nets. Moreover, calculation reveals that $N(H(\nu)) = \nu$, as required. ■

7.1.2 From SFC-nets to metagraphs

The significance of Proposition 7.1 lies in asserting that SFC-nets are "essentially the same" as a subset of graph homomorphisms into $\mathcal{N}$. In isolation, however, this is hardly a compelling reason for adopting such graph homomorphisms as models of SFCs. The advantage to be gained by such a change in perspective arises from the connection between graph homomorphisms and a particular notion of "typing". One particularly appealing way of thinking about the graph $\mathcal{N}$ is by analogy to an algebraic signature; the objects $\bullet$, $\square$ playing the role the "sorts" (or "types") and the arrows playing the role of "operator symbols". A homomorphism $h : G \to \mathcal{N}$ may then be regarded as a "typed term" which respects the typing constraints imposed by the signature: no composable arrows a, b in G may be both mapped to elements of $\mathbf{S}$ (or $\mathbf{T}$). This captures the strict alternation of steps and transitions in (valid) SFC programs. By introducing types explicitly in this way we have made the first step in developing an algebra of homomorphisms into $\mathcal{N}$, and thus and algebra of "typed SFC terms".

The situation just described is an instance of a general approach: the idea of using graphs as "generalised signatures" is central to the theory of *sketches* [8, 83, 144]. Sketches provide a general specification framework which is based on category theory and supersedes algebraic and language-based specifications.

The next step is to extend $\mathcal{N}$ to a graph $\mathcal{N}'$ such that homomorphisms into $\mathcal{N}'$ may be used to specify not only SFC-nets but also SFC diagrams (which, in addition, include branching elements such as "forks" and "joins"). One such extension results from taking $\mathcal{N}' = \mathcal{R}(\mathcal{N})$, the reflexive closure of $\mathcal{N}$. The subsequent development of the algebra, however, does not depend on the exact nature of $\mathcal{N}$. Thus, we can now generalise our discussion beyond what is strictly necessary for treating SFCs:

Definition 7.3. Let G be a graph. A *metagraph in G* is a homomorphism $M : S \to \mathcal{R}(G)$ of graphs. The graph S is called the *shape* of metagraph M and will be denoted ShM. The sources and sinks of a metagraph M are defined to be those of its shape graph. ■

In the previous chapter, metagraphs were introduced as labelled graphs. This is because every graph homomorphism from S to G may be regarded as a consistent labelling of the objects and arrows of S with those of G. In the case of metagraphs, the labelling graph G is additionally reflexive. This allows arrows with the same source and target to be labelled by the distinguished arrows in G. By convention, the labels corresponding to the distinguished arrows of G will be omitted in pictorial presentations. Thus, in this sense, the labelling may be regarded as "partial" on the arrows of S, the distinguished arrows being regarded as "non-labels".

7.2 Categories of metagraphs

Definition 7.4. Let M, M' be metagraphs in G. A *morphism* $m : M \to M'$ is a graph homomorphism $m : \mathrm{Sh}M \to \mathrm{Sh}M'$ such that $M' \circ m = M$, i.e. such that the diagram

$$\begin{tikzcd}
\mathrm{Sh}M \arrow[rr, "m"] \arrow[dr, "M"'] & & \mathrm{Sh}M' \arrow[dl, "M'"] \\
& \mathcal{R}(G) &
\end{tikzcd}$$

in **Graph** commutes. ■

Clearly, the identity homomorphism $\mathrm{Sh}M \to \mathrm{Sh}M$ is a morphism $1_M : M \to M$ such that $1_{M'} \circ m = m = m \circ 1_M$ for every morphism $m : M \to M'$ of metagraphs in the same graph G. Also, for every two morphisms $m : M \to M'$ and $m' : M' \to M''$ of metagraphs in G, the graph homomorphism $m' \circ m$ is a morphism $m' \circ m : M \to M''$ of metagraphs and $\circ$ is associative. Thus, for every graph G one obtains a category **Meta**$[G]$ having all metagraphs in G as objects and all morphisms of metagraphs in G as arrows.

Proposition 7.2. *Each* **Meta**$[G]$ *has coproducts.*

Proof. (Sketch) The coproduct $M + M'$ is given by the unique arrow $[M, M']$ in **Graph** making the following coproduct diagram commute:

$$\begin{tikzcd}
\mathrm{Sh}M \arrow[r, "\mathrm{inl}"] \arrow[dr, "M"'] & \mathrm{Sh}M + \mathrm{Sh}M' \arrow[d, "{[M,M']}"] & \mathrm{Sh}M' \arrow[l, "\mathrm{inr}"'] \arrow[dl, "M'"] \\
& \mathcal{R}(G) &
\end{tikzcd}$$

Thus $\mathrm{Sh}(M + M') = \mathrm{Sh}M + \mathrm{Sh}M'$. The associated injections $\mathrm{inl}_{M,M'}$ and $\mathrm{inr}_{M,M'}$ are those associated with the coproduct $\mathrm{Sh}M + \mathrm{Sh}M'$ in **Graph**. Given any metagraph M'' in G and morphisms $f : M \to M''$, $f' : M' \to M''$, the unique morphism $m : M + M' \to M''$ such that $m \circ \mathrm{inl}_{M,M'} = f$ and $m \circ \mathrm{inr}_{M,M'} = f'$ is given by the unique graph homomorphism making

$$\begin{tikzcd}
\mathrm{Sh}M \arrow[r, "\mathrm{inl}"] \arrow[dr, "f"'] & \mathrm{Sh}M + \mathrm{Sh}M' \arrow[d, "m"] & \mathrm{Sh}M' \arrow[l, "\mathrm{inr}"'] \arrow[dl, "f'"] \\
& \mathrm{Sh}M'' &
\end{tikzcd}$$

commute. ■

Proposition 7.3. *Each* **Meta**$[G]$ *has pushouts.*

Proof. Let $f : M \to M_1$ and $g : M \to M_2$ be arrows in **Meta**$[G]$, where M, M_1 and M_2 have shapes S, S_1 and S_2 respectively. Form $S' = S_1 \amalg_{f,g} S_2$ and consider the following situation in **Graph**:

$$\begin{tikzcd}
& S \arrow[dl, "f"'] \arrow[dr, "g"] & \\
S_1 \arrow[dr, "t_1"] \arrow[ddr, "M_1"'] & & S_2 \arrow[dl, "t_2"'] \arrow[ddl, "M_2"] \\
& S' & \\
& \mathcal{R}(G) &
\end{tikzcd}$$

where (S', t_1, t_2) is a pushout of f, g. Clearly, $M_1 \circ f = M = M_2 \circ g$ by the fact that f and g are morphisms of metagraphs. Since (S', t_1, t_2) is a pushout of f, g, there exists unique graph homomorphism $M' : S' \to \mathcal{R}(G)$ such that $M' \circ t_1 = M_1$ and $M' \circ t_2 = M_2$. It follows that there exists object M' and arrows $t_1 : M_1 \to M'$, $t_2 : M_2 \to M'$ in **Meta**$[G]$ such that $t_1 \circ f = t_2 \circ g$.

Consider now metagraph M'' in G of shape S'' and metagraph morphisms $r_1 : M_1 \to M''$, $r_2 : M_2 \to M''$ such that $r_1 \circ f = r_2 \circ g$. From the pushout of f, g in **Graph** one again obtains unique graph homomorphism $j : S' \to S''$ such that

$j \circ t_1 = r_1$ and $j \circ t_2 = r_2$ in **Graph**. Consider now the following diagram

$$\begin{tikzcd}
S_1 \arrow[r, "t_1"] \arrow[dr, "r_1"] \arrow[ddr, "M_2"'] & S' \arrow[d, "j"'] & S_2 \arrow[l, "t_2"'] \arrow[dl, "r_2"'] \arrow[ddl, "M_2"] \\
 & S'' \arrow[d, "M''"] & \\
 & \mathcal{R}(G) &
\end{tikzcd}$$

in which the commutativity of the individual triangles gives $(j \circ M'') \circ t_1 = M_1$ and $(j \circ M'') \circ t_2 = M_2$. Thus, $j \circ M''$ satisfies the defining property of M' and so one necessarily has $j \circ M'' = M'$. Hence, j is the required, unique metagraph morphism $j : M' \to M''$ such that $j \circ t_1 = r_1$ and $j \circ t_2 = r_2$. ■

7.3 The tensor category of concatenable metagraphs

The purpose of this section is to provide a rigorous definition to the algebra of concatenable metagraphs.

7.3.1 Concatenable metagraphs

Intuitively, a concatenable metagraph is a metagraph M together with a sequence of sources of M and a sequence of sinks of M:

Definition 7.5. A *concatenable metagraph in G* is a triple (M, s_0, s_1) where:

- M is a metagraph in G
- s_0, s_1 are sequences such that, either

  $$s_0 : [n] \to \mathrm{Src}M \quad \text{and} \quad s_1 : [m] \to \mathrm{Snk}M$$

  or

  $$s_0 : [n] \to \mathrm{Snk}M \quad \text{and} \quad s_1 : [m] \to \mathrm{Src}M$$

  for some $n, m \in \mathbb{N}$. ■

This definition presents a slightly generalised view of concatenable metagraphs than that presented in the previous chapter. In all concatenable metagraphs (M, s_0, s_1) arising in applications, s_0 will be the sequence of sources and s_1 the sequence of sinks. However, the formal definition refrains from making this commitment explicit. This is because we shall require the notion that each concatenable metagraph C has a *dual* $C^\star$:

Definition 7.6. $(M, s_0, s_1)^\star \overset{\text{def}}{=} (M, s_1, s_0)$. ■

It is now possible to provide a succinct account for the typing scheme introduced in Section 6.4.1.5.

Definition 7.7. Let $C = (M, s_0, s_1)$ be a concatenable metagraph in G with s_0 and s_1 being of length n and m respectively. The *type* of C is an ordered pair $\mathit{type}(C) \overset{\text{def}}{=} \langle \alpha, \beta \rangle$ of strings of objects in G, where $\alpha = a_1 \ldots a_n$, $\beta = b_1 \ldots b_m$ with $a_i = M(s_0(i))$ and $b_j = M(s_1(j))$. We shall write $C : \alpha \to \beta$ to assert that C is a concatenable metagraph of type $\langle \alpha, \beta \rangle$. ■

Clearly $C^\star : \beta \to \alpha$ if and only if $C : \alpha \to \beta$.

Given any string $\alpha = a_1 \ldots a_k$ of objects in G, the mapping $i \mapsto a_i$, $1 \leq i \leq k$ defines a metagraph in G of shape $([k], \emptyset, \dots)$ (i.e. the graph with objects $\{1, \dots, k\}$ and no arrows). This metagraph will also be denoted α.

Thus, given any concatenable metagraph $(M, s_0, s_1) : \alpha \to \beta$ in G one may conveniently regard the sequences s_0, s_1 also as metagraph morphisms $s_0 : \alpha \to M$ and $s_1 : \beta \to M$.

Definition 7.8. $(M, s_0, s_1) : \alpha \to \beta$ and $(M', s'_0, s'_1) : \alpha \to \beta$ in G are *isomorphic* if there exists an isomorphism $i : M \to M'$ of metagraphs such that $s'_0 = j \circ s_0$ and $s'_1 = j \circ s_1$. That is, the diagram

$$\begin{tikzcd}
 & M \arrow[dd, "j"] & \\
\alpha \arrow[ur, "s_0"] \arrow[dr, "s'_0"'] & & \beta \arrow[ul, "s_1"'] \arrow[dl, "s'_1"] \\
 & M' &
\end{tikzcd}$$

in **Meta**$[G]$ commutes. ■

It is reasonable to disregard differences in concatenable metagraphs arising solely from the nature of objects and arrows in their respective shape graphs. This is because all the information of interest lies in the homomorphism part of the definition, not the formal details of the shape graphs. From now on, isomorphic concatenable metagraphs will be identified (i.e. considered to be the same).

7.3.2 Operations on concatenable metagraphs

Assume given, and fixed in the rest of this section, a graph G. We are now in position to define a few basic operations among concatenable metagraphs in G.

7.3.2.1 Sequential Composition

Definition 7.9. The sequential composition of $C = (M, s_0, s_1) : \alpha \to \beta$ and $C' = (M', s'_0, s'_1) : \gamma \to \delta$ is only defined when $\beta = \gamma$, in which case it is denoted $C;C'$. Specifically,

$$C;C' \overset{\text{def}}{=} (M'',\ t \circ s_0,\ t' \circ s'_1) : \alpha \to \delta\ ,$$

where $(M'', t : M \to M'', t' : M' \to M'')$ is a pushout of s_1 and s'_0 in $\mathbf{Meta}[G]$. ■

Thus, the shape of $C;C'$ is (isomorphic to) $S \amalg_{s_1, s'_0} S'$ where S and S' are the shapes of C and C' respectively. Under the intended interpretation of the sequences, the construction of $\mathrm{Sh}(C;C')$ may be intuitively understood as the process of:

1. making two disjoint copies of S and S'; and

2. subsequently identifying each sink $s_1(i)$ in S with the source $s'_0(i)$ of S'.

M'' is then the metagraph of shape $\mathrm{Sh}(C;C')$ which agrees with M and M' on the constituent parts.

Remark. *Since any two pushouts of the same pair of arrows are isomorphic, the definition of* ; *does not depend on any particular choice for* (M, t, t')*: any two legitimate choices will result in concatenable metagraphs which are isomorphic and thus identical.*

Definition 7.10. Given string α of objects in G and of length n, let

$$1_\alpha \overset{\text{def}}{=} (\alpha,\ 1_{[n]},\ 1_{[n]}) : \alpha \to \alpha \quad ■$$

Proposition 7.4. *The following identities hold, whenever the composites involved are defined:*

1. $1_\alpha\,;C = C$ *and* $C\,;1_\beta = C$ *for all* $C : \alpha \to \beta$

2. $C_1\,;(C_2\,;C_3) = (C_1\,;C_2)\,;C_3$

Proof. Both parts should be intuitively clear. We shall, however, supply a rigorous proof of (2) in order to illustrate the use of the category-theoretic definitions.

Let $C_i = (M_i, s_{i0}, s_{i1})$ with $C_1 : \alpha \to \beta$, $C_2 : \beta \to \gamma$ and $C_3 : \gamma \to \delta$. Form

$$\begin{aligned} C_1;C_2 &= (M_{12},\ t \circ s_{10},\ t' \circ s_{21}) \\ C_2;C_3 &= (M_{23},\ r \circ s_{20},\ r' \circ s_{31}) \\ C_1;(C_2;C_3) &= (M,\ q \circ s_{10},\ q' \circ (r' \circ s_{31})) \end{aligned}$$

and a pushout of t', r:

$$\begin{tikzcd}
\alpha \arrow[dr, "s_{10}"] & & \beta \arrow[dl, "s_{11}"'] \arrow[dr, "s_{20}"] & & \gamma \arrow[dl, "s_{21}"'] \arrow[dr, "s_{30}"] & & \delta \arrow[dl, "s_{31}"'] \\
& M_1 \arrow[dr, "t"] \arrow[dddrr, "q"'] & (1) & M_2 \arrow[dl, "t'"'] \arrow[dr, "r"] & & M_3 \arrow[dl, "r'"'] & \\
& & M_{12} \arrow[dr, "w"] & (2) & M_{23} \arrow[dl, "w'"'] \arrow[ddl, "q'"] & & \\
& & & M' & & & \\
& & & M & & &
\end{tikzcd}$$

The squares marked (1) and (2) are both pushout squares. By Lemma 5.4 so is the square with sides s_{11}, $r \circ s_{20}$ and vertex M'. Thus, there is a unique isomorphism $i : M \to M'$ such that

$$i \circ q = w \circ t \ \text{ and } \ i \circ q' = w' \ .$$

(This is because the square with sides s_{11}, $r \circ s_{20}$, q and q' is also a pushout.) Similarly, one constructs

$$(C_1; C_2); C_3 = (M'', \ z \circ (t \circ s_{10}), \ z' \circ s_{31})$$

and shows the existence of unique isomorphism $j : M' \to M''$ such that

$$j \circ w = z \ \text{ and } \ j \circ w' \circ r' = z' \ .$$

It now follows that the isomorphism $j \circ i : M \to M''$ satisfies

$$(j \circ i) \circ (q \circ s_{10}) = j \circ w \circ t \circ s_{10} = z \circ t \circ s_{10}$$

and

$$(j \circ i) \circ (q' \circ r' \circ s_{31}) = j \circ w' \circ r' \circ s_{31} = z' \circ s_{31} \ .$$

This proves

$$\begin{aligned}
C_1; (C_2; C_3) \ &= \ (M, \ q \circ s_{10}, \ q' \circ (r' \circ s_{31})) \\
&= \ (M'', \ z \circ (t \circ s_{10}), \ z' \circ s_{31}) \\
&= \ (C_1; C_2); C_3 \ .
\end{aligned}$$

■

We have thus shown that every graph G gives rise to a category $\mathbf{CMeta}[G]$ in which:

- the objects are all finite strings of objects in G
- the arrows $C : \alpha \to \beta$ are all concatenable metagraphs in G of type $\langle \alpha, \beta \rangle$
- the identities are $1_\alpha : \alpha \to \alpha$
- the composition $C' \circ C$ for each $C : \alpha \to \beta$ and $C' : \beta \to \gamma$ is given by $C; C'$.

7.3.2.2 Parallel Composition

Definition 7.11. The parallel composition (or "disjoint union") of $C : \alpha \to \beta$ and $C' : \gamma \to \delta$, denoted $C \otimes C'$, is the concatenable metagraph

$$(M + M',\ s_0 + s_0',\ s_1 + s_1') : \alpha\gamma \to \beta\delta \ ,$$

where $M + M'$ is the coproduct of metagraphs in $\mathbf{Meta}[G]$, $s_i + s_j'$ is the coproduct of sequences (regarded as arrows in **Set**) and $\alpha\beta$ denotes the concatenation of strings α, β. ■

In the previous definition, we have exploited the bijection $[n+m] \cong [n] + [m]$ given by the mappings $x \mapsto \mathrm{inl}(x)$ and $y \mapsto \mathrm{inr}(y - n)$ for all $1 \leq x \leq n$ and $n + 1 \leq y \leq n + m$, to regard the coproduct of sequences $s : [n] \to A$ and $s' : [m] \to B$ as a sequence $s + s' : [n + m] \to A + B$. The same bijection provides an isomorphism of metagraphs $\alpha\beta \cong \alpha + \beta$, which we shall treat as interchangeable.

Proposition 7.5. *Let ϵ denote the empty string. The following identities hold, whenever the composites involved are defined:*

1. $1_\alpha \otimes 1_\beta = 1_{\alpha\beta}$
2. $C \otimes 1_\epsilon = 1_\epsilon \otimes C = C$
3. $C_1 \otimes (C_2 \otimes C_3) = (C_1 \otimes C_2) \otimes C_3$
4. $(C_1 \otimes C_2); (C_1' \otimes C_2') = (C_1; C_1') \otimes (C_2; C_2')$

Proof. (1), (2) and (3) are immediate consequences of Definition 7.11. For (4) let

$$\begin{aligned} C_i &= (M_i,\ s_{i0},\ s_{i1}) : \alpha_i \to \beta_i \\ C_i' &= (M_i',\ s_{i0}',\ s_{i1}') : \beta_i \to \gamma_i \\ C_i \,;\, C_i' &= (M_i'',\ t_i \circ s_{i0},\ t_i' \circ s_{i1}') \end{aligned}$$

$i = 1, 2$. Form

$$(C_1 \otimes C_2); (C'_1 \otimes C'_2) = (M,\ r \circ (s_{10} + s_{20}),\ r' \circ (s'_{11} + s'_{21}))$$

with associated pushout square:

$$\begin{tikzcd}
\beta_1 + \beta_2 \arrow[r, "s_{11}+s_{21}"] \arrow[d, "s'_{10}+s'_{20}"'] & M_1 + M_2 \arrow[d, "r"] \\
M'_1 + M'_2 \arrow[r, "r'"'] & M
\end{tikzcd}$$

From the pushout squares corresponding to the $C_i; C'_i$ and Lemma 5.5 one obtains that

$$\begin{tikzcd}
\beta_1 + \beta_2 \arrow[r, "s_{11}+s_{21}"] \arrow[d, "s'_{10}+s'_{20}"'] & M_1 + M_2 \arrow[d, "t_1+t_2"] \\
M'_1 + M'_2 \arrow[r, "t'_1+t'_2"'] & M''_1 + M''_2
\end{tikzcd}$$

is also a pushout square. Thus, there exists unique isomorphism $j : M \to M''_1 + M''_2$ such that $j \circ r = t_1 + t_2$ and $j \circ r' = t'_1 + t'_2$. Using Lemma 5.2, it follows that

$$j \circ r \circ (s_{10} + s_{20}) = (t_1 + t_2) \circ (s_{10} + s_{20}) = (t_1 \circ s_{10}) + (t_2 \circ s_{20})$$

and

$$j \circ r' \circ (s'_{11} + s'_{21}) = (t'_1 + t'_2) \circ (s'_{11} + s'_{21}) = (t'_1 \circ s'_{11}) + (t'_2 \circ s'_{21}) \ .$$

This proves

$$\begin{aligned}
&(C_1 \otimes C_2); (C'_1 \otimes C'_2) = \\
&\qquad (M,\ r \circ (s_{10} + s_{20}),\ r' \circ (s'_{11} + s'_{21})) = \\
&\qquad (M''_1 + M''_2,\ (t_1 \circ s_{10}) + (t_2 \circ s_{20}),\ (t'_1 \circ s'_{11}) + (t'_2 \circ s'_{21})) \\
&\qquad = (C_1; C'_1) \otimes (C_2; C'_2) \ .
\end{aligned}$$

■

The operation $\otimes$ extends to the objects of each **CMeta**$[G]$ as: $\alpha \otimes \beta \overset{\text{def}}{=} \alpha\beta$. We have thus shown that each $(\mathbf{CMeta}[G], \otimes, \epsilon)$ is a tensor category.

7.3.2.3 Symmetries

Definition 7.12. Given strings α, β of objects in G and of lengths n and m respectively,

$$\mathbf{c}_{\alpha,\beta} \overset{\text{def}}{=} (\alpha\beta,\ \mathrm{id}_{[n+m]},\ \pi_{n,m})$$

where $\pi_{n,m}$ is the symmetric permutation of Example 5.6. ■

Proposition 7.6. *The following identities hold, where $C : \alpha \to \gamma$, $C' : \beta \to \delta$:*

1. $\mathbf{c}_{\alpha,\beta} ; \mathbf{c}_{\beta,\alpha} = 1_{\alpha\beta}$
2. $\mathbf{c}_{\alpha\beta,\gamma} = (1_\alpha \otimes \mathbf{c}_{\beta,\gamma}) \,;\, (\mathbf{c}_{\alpha,\gamma} \otimes 1_\beta)$
3. $(C \otimes C'); \mathbf{c}_{\gamma,\delta} = \mathbf{c}_{\alpha,\beta}; (C' \otimes C)$

Proof. Routine from the definitions, using the fact that symmetric permutations are a symmetry for the tensor category of permutations (Example 5.5). ■

Thus, each tensor category **CMeta**$[G]$ is also symmetric.

7.3.2.4 Loops

The simple loop construction of Section 6.4.1.4 generalises easily: multiple loops can be created when there are more than one matching pairs of sinks and sources, i.e. there exists k such that the i-th sink shares the same label as the i-th source for $1 \leq i \leq k$. The construction of pair-wise identifying these matching sources and sinks becomes the trace on each category **CMeta**$[G]$.

In order to obtain a formal definition of this construction, we first show that each **CMeta**$[G]$ has units and counits (Definition 5.11).

Definition 7.13. Let α be a string of objects in G and let n be the length of α. Define the following concatenable metagraphs in G

$$\begin{aligned} \mu_\alpha &\overset{\text{def}}{=} (\alpha,\ \epsilon,\ \delta_n) : \epsilon \to \alpha\alpha \\ \eta_\alpha &\overset{\text{def}}{=} (\alpha,\ \delta_n,\ \epsilon) : \alpha\alpha \to \epsilon\ \ , \end{aligned}$$

where ϵ denotes the empty sequence and the sequence δ_n of length $n+n$ is given by: $\delta_n(x) = x$ for $1 \leq x \leq n$ and $\delta_n(x) = x - n$ for $n+1 \leq x \leq n+n$. ■

Example 7.2. $\mu_{\diamondsuit\spadesuit\diamondsuit} : \epsilon \to \diamondsuit\spadesuit\diamondsuit\diamondsuit\spadesuit\diamondsuit$ is

$$\diamondsuit_{1,4} \qquad \spadesuit_{2,5} \qquad \diamondsuit_{3,6}\ \ ,$$

where we indicate sequences using superscripts and subscripts. (So here, for instance, each object occurs twice in the sequence and is indexed by both i and $i+3$.) Correspondingly, $\eta_{\diamondsuit\spadesuit\diamondsuit} : \diamondsuit\spadesuit\diamondsuit\diamondsuit\spadesuit\diamondsuit \to \epsilon$ is

$$ {}^{1,4}\diamondsuit \qquad {}^{2,5}\spadesuit \qquad {}^{3,6}\diamondsuit \quad . $$

Now, for C_2 as in Example 6.6, $\mu_{\diamondsuit\spadesuit\diamondsuit} \,;\, (C_2 \otimes 1_{\diamondsuit\spadesuit\diamondsuit})$ is the concatenable metagraph:

$$\begin{tikzcd}
\diamondsuit_3 \arrow[r, "g"] & \spadesuit_1 \\
\spadesuit_4 \arrow[r, "f"] & \diamondsuit_2 \\
\diamondsuit_5 \arrow[ur, "k"'] &
\end{tikzcd}$$

■

Proposition 7.7. *For every object α of* **CMeta**$[G]$*, $\langle \mu_\alpha, \eta_\alpha \rangle$ is a unit-counit pair.*

Proof. (Sketch) We outline the steps in verifying the four conditions in Definition 5.11:

1. $(\mu_\alpha \otimes 1_\alpha) \,;\, (1_\alpha \otimes \eta_\alpha) = 1_\alpha$. Let the length of α be n. One constructs

   $$(\mu_\alpha \otimes 1_\alpha) \,;\, (1_\alpha \otimes \eta_\alpha) = (\alpha,\ \delta_n \circ (\epsilon + 1_{[n]}),\ \delta_n \circ (1_{[n]} + \epsilon)) \ .$$

   It is then easy to see that $\delta_n \circ (\epsilon + 1_{[n]}) = 1_{[n]}$ and that $\delta_n \circ (1_{[n]} + \epsilon) = 1_{[n]}$.

2. $(1_\alpha \otimes \mu_\alpha) \,;\, (\eta_\alpha \otimes 1_\alpha) = 1_\alpha$. Similar to (1) above.

3. $\mu_\alpha \,;\, (C \otimes 1_\alpha) = \eta_\beta \,;\, (1_\beta \otimes C^\star)$ for all $C = (M, s_0, s_1) : \alpha \to \beta$. One begins by showing that

   $$\mu_\alpha \,;\, (C \otimes 1_\alpha) = (M,\ \epsilon,\ [s_1, s_0]) \ ,$$

   where $[s_1, s_0] : \beta + \alpha \to M$ is the unique metagraph morphism such that $[s_1, s_0] \circ \mathrm{inl} = s_1$ and $[s_1, s_0] \circ \mathrm{inr} = s_0$. (Intuitively, this should be clear. That it is formally a consequence of the definitions, however, is a rather laborious exercise.) Recalling that $C^\star = (M, s_1, s_0)$ one also shows that

   $$\eta_\beta \,;\, (1_\beta \otimes C^\star) = (M,\ \epsilon,\ [s_1, s_0]) \ .$$

4. $(C^\star \otimes 1_\alpha) \,;\, \eta_\alpha = (1_\beta \otimes C) \,;\, \eta_\beta$ for all $C = (M, s_0, s_1) : \alpha \to \beta$. Similar to (3) above.

■

We are now ready to define the general form $\circlearrowright C$:

Definition 7.14. Given sequences α, β, γ of G, $\circlearrowright^{\alpha}_{\beta,\gamma} C$ is the concatenable metagraph in G given by:

$$\circlearrowright^{\alpha}_{\beta,\gamma} C \overset{\text{def}}{=} (\mu_\alpha \otimes 1_\beta)\,;\,(1_\alpha \otimes C)\,;\,(\eta_\alpha \otimes 1_\beta)\ .$$

As a corollary of this definition and Proposition 5.3, the general form of the equational laws for $\circlearrowright^{\alpha}_{\beta,\gamma} C$ are now as follows:

- Vanishing: $\circlearrowright^{\epsilon}_{\alpha,\beta} C = C$, for $C : \alpha \to \beta$, and $\circlearrowright^{\gamma\delta}_{\alpha,\beta} C = \circlearrowright^{\delta}_{\alpha,\beta} (\circlearrowright^{\gamma}_{\delta\alpha,\delta\beta} C)$, for $C : \gamma\delta\alpha \to \gamma\delta\beta$
- Superposing: $\circlearrowright^{\alpha}_{\beta\delta,\gamma\delta} (C \otimes 1_\delta) = (\circlearrowright^{\alpha}_{\beta,\gamma} C) \otimes 1_\delta$, for $C : \alpha\beta \to \alpha\gamma$.
- Yanking: $\circlearrowright^{\alpha}_{\alpha,\alpha} \mathbf{c}_{\alpha,\alpha} = 1_\alpha$
- Left-Tightening: $\circlearrowright^{\alpha}_{\beta,\delta} ((1_\alpha \otimes C'); C) = C'; (\circlearrowright^{\alpha}_{\gamma,\delta} C)$, $C : \alpha\gamma \to \alpha\delta, C' : \beta \to \gamma$
- Right-Tightening: $\circlearrowright^{\alpha}_{\beta,\delta} (C; (1_\alpha \otimes C')) = (\circlearrowright^{\alpha}_{\beta,\gamma} C)\,; C'$, $C : \alpha\beta \to \alpha\gamma$, $C' : \gamma \to \delta$
- Sliding: $\circlearrowright^{\alpha}_{\beta,\delta} (C\,; (C' \otimes 1_\delta)) = \circlearrowright^{\gamma}_{\beta,\delta} ((C' \otimes 1_\beta)\,; C)$, $C : \alpha\beta \to \gamma\delta, C' : \gamma \to \alpha$.

Thus each **CMeta**$[G]$ is traced.

7.4 Discussion

Our attempts to define an algebra associated with SFC diagrams led us to consider various approaches to the algebraic theory of Petri nets. One influential such approach is due to Meseguer and Montanari [100]. There, a net is regarded as a graph N whose objects are the places of the net (more generally, multisets thereof) and whose arrows are the transitions. Every such graph can be freely completed to a tensor category $T(N)$. Although this was already close to our objective, it suffers from a certain kind of asymmetry: arrow expressions over $T(N)$ suppress any explicit mention of places. This reflects an asymmetry in the semantics of nets which attaches computational significance only to the transitions. Our concern, instead, was to provide a syntactic description of nets which gives places and transitions equal status (as in Definition 7.1).

Nevertheless, the idea of concatenable metagraphs owes its very existence to work in [27, 120] extending the basic theory of net computations. There, a notion of "concatenable process" was introduced for nets. Suitably adapted, this inspired our Definition 7.5 of concatenable metagraphs.

Another algebraic view of nets, closer to ours, is the one of Katis, Sabadini and Walters [79]. There, nets are associated to arrows in Span(Graph), a tensor category constructed from **Graph**, in a way which maps both places and transitions to arrows in the category. Here one meets the other extreme: a "place-arrow" may share the same source and target as a "transition arrow". Owing to this lack of a formal distinction between arrows standing for places and those standing for transitions, one may construct expressions in the category which have no meaning as nets. Defining which expressions are meaningful in this approach appears to require an appeal to notational convention. (For the purposes of [79], however, this is not a problem.)

By contrast, our approach captures the typing constraints of SFC diagrams in the graph $\mathcal{N}$ over which our metagraphs are defined. This is done in a way which gives "step arrows" and "transition arrows" equal status, making in them both visible in the resulting expressions, from which one easily "reads off" the layout of SFC diagrams.

Chapter 8

Conclusions

The two main areas to which this thesis contributes are:

- *Research in the design and application of domain-specific programming languages*: Recent approaches in this area have regarded expressive power, rather than modality or form of syntax, as the primary distinction between domain-specific and general-purpose languages. This view can be accounted as the result of an almost exclusive focus on domains within mainstream computing, in which linguistic modalities of expression have been the definite norm. By contrast we have emphasised languages which, owing to their origins in engineering domains, use diagrams as the primary modality for expressing programs. However, our choice of emphasis represents more than mere curiosity. It reflects our conviction that diagrams are a very natural syntax for software in certain domains; particularly those in which diagrammatic notations form an integral part of design practice and technical discourse.

- *The study of diagrammatic notations and their role in reasoning*: In this area we have contributed a study in the *computational* interpretation of software diagrams and of the consistency issues that such interpretations raise. Accordingly, we have addressed how the structure of diagrams may be exploited in guiding reasoning about genuinely computational properties of systems, such as safety. By contrast, previous studies of diagrammatic reasoning have focused on notations for syllogistic or set-theoretic inference, e.g. Euler circles and Venn diagrams. For such applications, simple model-theoretic structures suffice, whereas our need to address *dynamic behaviour* has demanded sophisticated semantic structures, such as transition systems and computational traces. In relating diagrams to their computational behaviour, we have argued for the merits of compositionality. The latter notion is fundamentally linked to *algebraic* methods of diagram description, which we argued to be superior analytic tools over grammatical approaches.

8.1 Summary of main results

This thesis has identified several issues arising from the use of diagrammatic notations as syntax in domain-specific programming languages. For each issue identified, we have argued for its importance, subjected it to mathematical analysis and derived methodological advice to the extent in which similar issues may be dealt with in the design of future languages.

We have used two examples of diagrammatic languages, drawn from the domain of embedded and industrial process control, as a basis for our case studies. The first language, known as “Function Blocks”, comprises box-and-wire diagrams presenting a data-flow view of an embedded controller. The second, known as “Sequential Function Charts” (SFCs), utilises variants of Petri net diagrams to specify the control flow of programs.

The first issue of concern regards the degree of correspondence between the structure of a diagram and the semantic structure capturing the diagram’s computational behaviour. We have argued that the representation offered by the diagrams must be homomorphic and systematic with respect to the represented semantic structures. Applying a concept due to Gurr, we have demonstrated how certain SFC diagrams fail this criterion, thereby severely reducing their capacity to support reasoning.

The class of system properties for which reasoning support is required was identified as a major characteristic of many domains. We outlined a language design methodology which emphasises ease of proof for such domain-characteristic properties. A concrete illustration of our approach was presented by developing a semantics and compositional proof system for SFC diagrams and simple safety properties.

Important consistency issues arise from the coexistence, within the same language or programming environment, of both conventional (i.e. textual) and diagrammatic means for expressing programs. Such, indeed, is the case with function blocks. We have developed a formal criterion ensuring consistence, formulated in terms of two algebras sharing the same signature: a term algebra T and an algebra D whose elements are models of the diagrams. A collection E of equations is imposed, capturing one’s ability to structurally decompose each given composite diagram in more than one way. To ensure consistency, one is required to show D to be the free algebra over the signature and E, thus showing D isomorphic to a quotient of T, and, moreover, that the equations in E are respected by the semantic interpretation of the terms in T. A complete, detailed application of this general criterion to the language of function blocks was presented.

In an attempt to address some of the pragmatic issues involved in the use of diagrammatic syntax we concentrated on layout and its role in structuring reasoning arguments. We investigated how the recent development of traced categories may be used to develop algebraic descriptions of graph-based diagrams which supply both structural and layout information. In this approach, one constructs an algebra of expressions denoting arrows in a traced category, the axioms in the category capturing equivalence of diagrams under layout transformation. We developed such an algebra capturing the basic layout of SFC diagrams and indicated how it may form the basis of a formal reasoning system. The application rests on a result relating the basic structure of SFC diagrams to a certain class of graph homomorphisms, which we call metagraphs, and the construction of an appropriate category thereof.

8.2 Directions for further work

The main open question raised by this thesis regards the existence of a metatheory for domain-specific languages. By this we mean both a cognitive analysis of their use in practice and a collection of mathematical models and results which underpin their design and associated reasoning. We envisage four main directions in which our work may be extended:

1. an empirical study of domain-specific notations and how they support reasoning about systems;

2. an analysis of the representations employed by designers, using a combined cognitive/computational account of how well these support typical reasoning tasks;

3. the development of an algebraic theory of representations whose adequacy will be judged by its ability to support features identified in (2) above;

4. the formalisation of reasoning tasks and the extent to which the designers' reasoning styles can be supported by formal systems.

8.2.1 Empirical work

Empirical observation aims to develop a sharp characterisation of the role of notation in supporting domain practices and to gather concrete examples of its use. The objective should be to gather typical examples of reasoning practice in an industrial context rather than to generate a definitive study of industrial design practice. Two possible activities in such a study would be:

- The *documentation* of notations used in industrial and engineering domains and of how they support common practices and tasks.
- The *abstraction* of the data collected to create a number of scenarios that cover as many aspects of notations and their use as possible.

Particular emphasis should be placed on how practitioners believe notation contributes to limiting variability in designs and explicating key properties of artifacts.

8.2.2 Cognitive analysis of representations

The objective here is to determine which cognitive properties a system representation must exhibit to be effective in its target domain. Effectiveness in this setting refers to the degree in which representations aid understanding of systems and support user tasks, and is measured in terms of closeness of match between representation and system/task.

Previous studies of diagrammatic representations have typically sought to explicate either computational benefits of diagrams through analysis of inherent constraints [9, 60, 123, 124, 125]; or (from an HCI perspective) features and properties which impact upon the cognition of the user [39, 43, 85, 109, 129]. Studies such as [41, 130, 148] have indicated that the most effective representations are those which are well matched to what they represent, in the context of particular reasoning tasks. Furthermore, it has been demonstrated in [43, 110, 112] that pragmatic features of diagrammatic representations (termed “secondary notations” by Green [42]) play a significant role in achieving such matching.

Earlier work at Edinburgh [47], informed by these previous analyses and the results of psychological studies of individual differences in users responses to various representations and reasoning tasks [105, 128], explored in general terms the issues that determine both computational and cognitive effectiveness of diagrammatic representations. Subsequently, this general exploration was refined in [50] to focus on those issues which determine whether a representation is well-matched, and how this matching is achieved. Initial applications of the informal framework suggested by this work to domain-specific representations are very promising [4, 48].

We aim to continue the refinement of this approach through the formalisation of the framework sketched in [50], leading to a principled approach for constructing ontological models of diagrammatic representations which identify their key properties and their syntactic and pragmatic features. The application of this framework to diagrams in general is problematic, as the assorted computational and cognitive factors vary significantly between diagrammatic languages. Furthermore, individual differences can substantially disturb the effects of representations on reasoning tasks. We believe that focusing on designers using domain-specific languages simplifies this problem, by reducing the variation in these assorted factors.

8.2.3 A mathematical theory of representation

A mathematical characterisation is sought of how representations relate to each other and to the concrete systems they stand for.

- *Equivalences and morphisms of diagrams.* Any given system may be represented in a variety of—subtly or grossly—different ways. In particular, diagrammatic representations may be decomposed (and thus semantically interpreted) in multiple ways. It is important for users in a domain to know exactly which such differences are significant, and which are not. Algebra and category theory provide rich frameworks in which precise notions of equivalence and morphism between diagrams may be formulated and studied.

On the whole, the methods of category theory seem highly pertinent to the study of diagrams. Moreover, a seamless extension of familiar algebraic techniques from sets to categories has recently been achieved in the work of Power and others [116, 118, 82]. Subject to adaptation, we expect their novel concept of *sketch* [83] to provide a useful device in the conceptual understanding of models of diagrams and their morphisms.

- *Representations and Behaviour.* Successful representation formalisms often promote a conceptual understanding of systems which is considerably removed from the mechanics of actual system operation. Unless the relation between representation and intended system behaviour is made precise, certain permitted implementations might behave differently to what the users in the domain expect [4]. Also, it is important to guarantee that equivalent representations result in equivalent behaviour. In this thesis, such issues have been dealt with in the context of specific diagrammatic languages and their semantics. Much further work is required before one obtains more general results linking static representations with their dynamic computational interpretations.

8.2.4 Supporting domain-specific reasoning

This direction considers the practical applications of the theory to the problem of supporting designers in providing evidence of desirable properties in systems.

While the use of formal methods in supplying evidence is often recommended or mandated (e.g. SEMSPLC guidelines [72], MOD-00-55 standard), their practical application remains undeniably difficult, relying on intimate knowledge and explicit manipulation of some underlying, generic model. By contrast, many informal or semi-formal arguments are guided by the structure of some high-level representation of the system, such as a diagram. In the course of such arguments, the representation itself is often manipulated in order to expose particular aspects which seem most pertinent to the goal. Nevertheless, the elevation of such arguments to a level admissible as rigorous evidence requires their formal *underpinning, validation* and *justification* by logical means. Viewing, as we do, formalisation as a means to the validation of informal (or semi-formal) reasoning practices emphasises mathematical analysis as a tool of the *notation (language) designer*, not as an imposition on the user.

Thus, our aim is to apply the theory connecting diagrammatic representations to semantics in seeking criteria under which representation-driven arguments are deemed sufficient to provide *rigorous* evidence without the need of explicitly constructing fully formal proofs. Doing so successfully seems unlikely in the absence of any restrictions regarding the class of diagrams and semantics under consideration. Also, the strictness of the criteria is likely to vary considerably across domains. In generalising from the case studies presented in this thesis, one must still restrict attention to fairly circumscribed classes of diagrams, such as those sharing some common, underlying graph structure.

Bibliography

[1] R. Alur and D. Dill. Automata for modelling real-time systems. In *Proceedings of ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] S. Anderson and K. Tourlas. A language for programmable controllers. Unpublished (available from the authors), 1997. Notes detailing an operational semantics for IEC 1131-3 function blocks.

[4] Stuart Anderson and Konstantinos Tourlas. Diagrams and programming languages for programmable controllers. In *Proceedings of the Formal Methods Europe Symposium*, volume 1313 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 1997.

[5] Stuart Anderson and Konstantinos Tourlas. Design for proof: An approach to the design of domain-specific languages. In J. F. Groote, B. Luttik, and J. van Wamel, editors, *Proceedings of the Third International Workshop on Formal Methods for Industrial Critical Systems*, pages 1–16. ERCIM, 1998.

[6] Stuart Anderson and Konstantinos Tourlas. Design for proof: An approach to the design of domain-specific languages. *Formal Aspects of Computing*, 10:452–468, 1998.

[7] G. Arango and R. Prieto-Diaz. Domain analysis concepts and research direction. In *Domain Analysis and Software Systems Modeling*, pages 9–31. IEEE Computer Society Press, 1991.

[8] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[9] J Barwise and J Etchemendy. Heterogeneous logic. In J Glasgow, N H Narayan, and B Chandrasekaran, editors, *Diagrammatic Reasoning: Cognitive and Computational Perspectives*, pages 211–234. MIT Press, 1995.

[10] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[11] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-flow synchronous languages. In *A Decade of Concurrency — Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[12] Albert Benveniste, Paul Le Guernic, Yves Sorel, and Michel Sorine. A denotational theory of synchronous reactive systems. *Information and Computation*, 99(2):192–230, August 1992.

[13] Gérard Berry. Real-time programming: Special purpose or general purpose languages. *Information Processing*, pages 11–17, 1989.

[14] Gérard Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. 1998.

[15] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):82–152, 1992.

[16] Frédéric Boussinot and Robert de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[17] David Bruce. What makes a good domain specific language? In *Proceedings of the First ACM Workshop on Domain-Specific Languages*. ACM, 1997.

[18] Luca Cardelli. Service combinators for Web computing. In Ramming [117], pages 1–10.

[19] E. Carlson, P. Hudak, and M. Jones. An experiment using Haskel to prototype "geometric reagion servers" for navy command and control. Technical Report YALEU/DCS/RR-1031, Department of Computer Science, Yale University, 1994.

[20] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 178–188. ACM, 1987.

[21] Paul Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[22] F. Cassez. Formal semantics of reactive Grafcet. *European Journal of Automation*, 31(3), 1997.

[23] International Electrotechnical Commission. *International Standard 1131, Programmable Controllers. Part 3: Programming Languages.* IEC, 1993.

[24] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation.* World Scientific, 1997.

[25] B Courcelle. Graph rewriting: An algebraic and logical approach. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier, 1990.

[26] R. Crole. *Categories for Types.* Cambridge Mathematical Textbooks, 1993.

[27] Pierpaolo Degano, José Meseguer, and Ugo Montanari. Axiomatizing the algebra of net computations and processes. *Acta Informatica*, 33:641–667, 1996.

[28] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Proceedings on the First International Workshop on Graph Grammars*, number 73 in Lecture Notes in Computer Science, pages 1–69. Springer-Verlag, 1979.

[29] Conal Elliott. Modelling interactive 3D and multimedia animation with an embedded language. In Ramming [117].

[30] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of SIGPLAN ICFP, International Conference on Functional Programming.* ACM, 1997.

[31] Ryszard Engelking. *General topology*, volume 6 of *Sigma series in pure mathematics.* Springer-Verlag, 1989.

[32] Dawson Engler. Incorporating application semantics and control into compilation. In Ramming [117], pages 103–115.

[33] Martin Erwig. Abstract visual syntax. In *IEEE Workshop on the Theory of Visual Languages*, 1997.

[34] Markus Fromherz, Vineet Gupta, and Vijay Saraswat. cc — a generic framework for domain specific languages. In Kamin [76].

[35] Markus Fromherz and Vijay Saraswat. Model-based computing: Using concurrent constraint programing for modeling and model compilation. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming - CP'95*, number 97 in LNCS. Springer-Verlag, 1995.

[36] P. Pepper G. Egger, A. Fett. Formal specification of a safe PLC language and its compiler. In *Proceedings of the SafeComp'94, 13th International Conference on Computer Safety, Reliability and Security.* ISA, 1994.

[37] Phillipa Gardner. Graphical presentations of interactive systems. Paper associated with the Mathfit Summer School at Imperial College, London, 1998.

[38] Thierry Gautier and Paul Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer-Verlag, 1987.

[39] J Glasgow, N H Narayan, and B Chandrasekaran. *Diagrammatic Reasoning: Cognitive and Computational Perspectives.* MIT Press, 1995.

[40] J. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

[41] J Good. VPLs and novice program comprehension: How do different languages compare? In *15th IEEE Symposium on Visual Languages (VL'99)*, pages 262–269. IEEE Computer Society, 1999.

[42] T R G Green. Cognitive dimensions of notations. In A Sutcliffe and Macaulay, editors, *People and Computers V*, pages 443–460. Cambridge University Press, 1989.

[43] T R G Green and M Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Visual Languages and Computing*, 7:131–174, 1996.

[44] D. Gries and F. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[45] C. Gunter, J. Mitchell, and D. Notkin. Strategic directions in software engineering and programming languages. *ACM Computing Surveys*, 28(4):727–737, December 1996.

[46] C Gurr. On the isomorphism (or otherwise) of representations. In *International Workshop on Theory of Visual Languages, held in Conjunction with AVI'96*, Gubbio, Italy, May 1996.

[47] C Gurr, J Lee, and K Stenning. Theories of diagrammatic reasoning: distinguishing component problems. *Mind and Machines*, 8(4):533–557, December 1998.

[48] C. Gurr and K. Tourlas. Formalising pragmatic features of graph-based notations. In *15th IEEE Symposium on Visual Languages (VL'99)*, pages 220–227. IEEE Computer Society, 1999.

[49] C. Gurr and K. Tourlas. Towards the principled design of software engineering diagrams. In *Proceedings of Int. Conf. On Software Engineering*, 2000. To appear.

[50] C A Gurr. Effective diagrammatic communication: Syntactic, semantic and pragmatic issues. *Journal of Visual Languages and Computing*, 10(4):317–342, August 1999.

[51] Corin Gurr. Theories of visual and diagrammatic reasoning: Foundational issues. In G. Allwein, K. Marriot, and B. Meyer, editors, *Formalizing Reasoning with Visual and Diagrammatic Representations*, AAAI Fall Symposium, 1998.

[52] Corin A. Gurr. On the isomporphism of representations. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 288–301. Springer-Verlag, 1998.

[53] Douglas Gurr. *Semantic Frameworks for Complexity*. PhD thesis, Department of Computer Science, University of Edinburgh, 1990.

[54] Wolfgang Halang, Soon-Hey Jung, Bernd Krämer, and Johan Scheepstra. *A Safety Licensable Computing Architecture*. World Scientific, 1993.

[55] Wolfgang Halang, Bernd Krämer, and Norbert Völker. Formally verified firmware modules for industrial process automation. In G. Rabe, editor, *14th International Conference on Computer Safety, Reliablity and Security (SAFECOMP'95)*, pages 206–218, 1995.

[56] Wolfgang A. Halang and Alceu Heinke Frigeri. Methods and languages for safety related real time programming. In *17th International Conference on Computer Safety, Reliablity and Security (SAFECOMP'98)*, 1998.

[57] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[58] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[59] A. Hamilton. *Logic for Mathematicians*. Cambridge University Press, 1978.

[60] E Hammer and N Danner. Towards a model theory of Venn diagrams. In J Barwise and G Allwein, editors, *Logical Reasoning with Diagrams*, pages 109–127. Oxford University Press, New York, 1996.

[61] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, volume 13 of *NATO ASI Series F*. Springer-Verlag, 1985.

[62] David Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3):231–275, 1987.

[63] Masahito Hasegawa. *Models of Sharing Graphs: Categorical Semantics of* `let` *and* `letrec`. PhD thesis, Department of Computer Science, University of Edinburgh, 1997.

[64] Massahito Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *Proceedings of Third International Conference on Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 196–213. Springer-Verlag, 1997.

[65] Eric Hehner. Abstractions of time. In *A Classical Mind: Essays in Honour of C.A.R. Hoare*, International Series in Computer Science. Prentice Hall, 1994.

[66] T. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[67] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112(2):237–337, 1994.

[68] Thomas Troels Hildebrandt, Prakash Panangaden, and Glynn Winskel. A relational model of non-deterministic dataflow. In *Proceedings of CONCUR'98*, 1998.

[69] Boudewijn Hoogeboom and Wolfgang Halang. The concept of time in the specification of real time systems. In *Real-Time Systems*, pages 19–38. IEEE Computer Society Press, 1992.

[70] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), December 1996.

[71] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of the International Conference on Software Reuse*, 1998.

[72] Institution of Electrical Engineers. *SEMSPLC Guidelines: Safety-Related Application Software for Programmable Logic Controllers*, volume 8 of *IEE Technical Guidelines*. IEE, London, 1996.

[73] Alan Jeffrey. Premonoidal categories and a graphical view of programs. Technical report, School of Cognitive and Computing Sciences, University of Sussex, 1997. Available as `http://klee.cs.depaul.edu/premon`.

[74] A. Joyal and R. Street. The geometry of tensor calculus I. *Advances in Mathematics*, 88:55–112, 1991.

[75] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

[76] S. Kamin, editor. *Proceedings of the First ACM Workshop on Domain-Specific Languages*. ACM, 1997.

[77] Samuel Kamin. Moving functional languages into the real world. In *Proceedings of Joint Brazilian/US Workshop on Formal Foundations of Software Systems*, 1997.

[78] Samuel Kamin and David Hyatty. A special-purpose language for picture drawing. In Ramming [117], pages 297–306.

[79] P. Katis, N. Sabadini, and R. F. C. Walters. Representing Place/Transition nets in Span(Graph). In Michael Johnson, editor, *Proceedings of the Sixth AMAST Conference*, volume 1349 of *Lecture Notes in Computer Science*, pages 323–336. Springer-Verlag, 1997.

[80] P. Katis, N. Sabadini, and R.F.C. Walters. Bicategories of processes. *Journal of Pure and Applied Algebra*, 115:141–178, 1997.

[81] G. Kelly and M. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193–213, 1980.

[82] G. M. Kelly and A. J. Power. Adjunctions whose counits are coequalisers and presentations of finitary enriched monads. *Journal of Pure and Applied Algebra*, 89:163–179, 1993.

[83] Yoshiki Kinoshita, John Power, and Makoto Takeyama. Sketches. *Electronic Notes in Theoretical Computer Science*, 6, 1997.

[84] Bernd Krämer and Norbert Völker. A highly dependable computing architecture for safety-critical control applications. *Real-Time Systems Journal*, 1997.

[85] J H Larkin and H A Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11:65–99, 1987.

[86] R. W. Lewis. *Programming industrial control systems using IEC 1131-3.* Control Engineering Series. The Institution of Electrical Engineers (IEE), London, 1995.

[87] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22th Symposium on Principles of Programming Languages*. ACM, 1995.

[88] Saunders MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[89] O. Maffeïs, M. Morley, and A. Poigné. The synchronous approach to designing reactive systems. *Formal Methods in System Design*, 12(2):163–187, 1998.

[90] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume Specification. Springer-Verlag, 1992.

[91] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Workshop on Visual Languages*, 1991.

[92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *Proceedings of CONCUR'92*. Springer Verlag, LNCS 630, August 1992.

[93] L. Marce and P. Le Parc. Defining the semantics of languages for programmable controllers with the help of synchronous processes. *Control Engineering Practice*, 1(1), 1993.

[94] J Marks and E Reiter. Avoiding unwanted conversational implicature in text and graphics. In *Proceedings of AAAI-90*, pages 450–456, 1990.

[95] K Marriot, B Meyer, and K Wittenberg. A survey of visual language specification and recognition. In K Marriot and B Meyer, editors, *Visual Language Theory*. Springer Verlag, June 1998.

[96] Kim Marriott and Bernd Meyer. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, 8(4):374–402, 1997.

[97] Kim Marriott and Bernd Meyer. The CCMG visual language hierarchy. In Marriott and Meyer, editors, *Visual Language Theory*. Springer-Verlag, 1998.

[98] John McDermid and Keith Bennett. Software engineering research in the UK: A critical appraisal. Engineering Board Meeting EB/25/98, The British Computer Society, November 1998.

[99] K. Meinke and J. V. Tucker. Universal algebra. In *Handbook of Logic In Computer Science*, volume 1, pages 189–397. Oxford Science Publications, 1995.

[100] José Meseguer and Ugo Montanari. Petri nets are monoids: A new algebraic foundation for net theory. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS)*, pages 142–154. IEEE Computer Society Press, 1988.

[101] Alex Mifsud. *Control Structures.* PhD thesis, Department of Computer Science, University of Edinburgh, 1996.

[102] Robin Milner. Flowgraphs and flow algebras. *Journal of the ACM*, 26(4):794–818, October 1979.

[103] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[104] Robin Milner. The polyadic pi-calulus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[105] P Monaghan and K Stenning. Effects of representational modality and thinking style on learning to solve reasoning problems. In *Proceeding of the 20th Annual Meeting of the Cognitive Science Society of America*, 1998.

[106] Matthew J. Morley. Safety-level communication in railway interlockings. *Science of Computer Programming*, 29(1–2):147–170, July 1997.

[107] Matthew John Morley. *Safety Assurance in Interlocking Design.* PhD thesis, Department of Computer Science, the University of Edinburgh, 1996.

[108] Lloyd Nakatani and Mark Jones. Jargons and infocentrism. In *Proceedings of the First ACM Workshop on Domain-Specific Languages*, 1997.

[109] N. H. Narayan. Diagrammatic communication: A taxonomic overview. In B Kokinov, editor, *Perspectives on Cognitive Science*, volume 3, pages 91–122. New Bulgarian University Press, Sofia, Bulgaria, 1997.

[110] J Oberlander. Grice for graphics: pragmatic implicature in network diagrams. *Information Design Journal*, 8(2):163–179, 1996.

[111] E. Parr. *Programmable Controllers, An Engineer's Guide.* Newnes, 1993.

[112] M Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–45, June 1995.

[113] M Petre and T R G Green. Requirements of graphical notations for professional users: electronics CAD systems as a case study. *Le Travail Humain*, 55:47–70, 1992.

[114] Benjamin Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[115] Axel Poigné. Basic category theory. In *Handbook of Logic In Computer Science*, volume 1, pages 416–634. Oxford Science Publications, 1995.

[116] John Power. Categories with algebraic structure. In Nielsen and Thomas, editors, *Proceedings of CSL 97*, volume 1414 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag, 1998.

[117] C. Ramming, editor. *Proceedings of the First USENIX Conference on Domain-Specific Languages*. USENIX, 1997.

[118] Edmund Robinson. Variations on algebra: Monadicity and generalisations of equational theories. Technical Report 1996:06, School of Cognitive and Computing Sciences, University of Sussex, 1994.

[119] D. E. Rydeheard and R. M. Burstall. Monads and theories: a survey for computation. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*. Cambridge, 1985.

[120] Vladimiro Sassone. An axiomatization of the category of Petri net computations. *Mathematical Structures in Computer Science*, 8(2):117–152, April 1998.

[121] David Schmidt. *Denotational Semantics, A Methodology for Language Development*. Wm. C. Brown Publishers, 1986.

[122] T. Sheard, Z. Banaissa, and E. Pasalic. DSL implementation using staging and monads. In *Proceedings of the Second USENIX Conference on Domain-Specific Languages*, 1999.

[123] A Shimojima. Operational constraints in diagrammatic reasoning. In J Barwise and G Allwein, editors, *Logical Reasoning with Diagrams*, pages 27–48. Oxford University Press, New York, 1996.

[124] A Shimojima. Derivative meaning in graphical representations. In *15th IEEE Symposium on Visual Languages (VL'99)*, pages 212–219. IEEE Computer Society, 1999.

[125] S-J Shin. Situation-theoretic account of valid reasoning with Venn diagrams. In J Barwise and G Allwein, editors, *Logical Reasoning with Diagrams*, pages 81–108. Oxford University Press, New York, 1996.

[126] Mei Chee Shum. Tortile tensor categories. *Journal of Pure and Applied Algebra*, 93:57–110, 1994.

[127] G. Stefanescu. Algebra of flownomials. Technical Report TUM-I9437, Technical University Munich, 1994.

[128] K Stenning, R Cox, and J Oberlander. Contrasting the cognitive effects of graphical and sentential logic teaching: reasoning, representation and individual differences. *Language and Cognitive Processes*, 10, 1995.

[129] K Stenning and J Oberlander. A cognitive theory of graphical and linguistic reasoning: logic and implementation. *Cognitive Science*, 19:97–140, 1995.

[130] K Stenning and P Yule. Image and language in human reasoning: a syllogistic illustration. *Cognitive Psychology*, 34(2):109–159, 1997.

[131] C. Stirling. Modal and temporal logics. In *Handbook of Logic In Computer Science*, volume 2, pages 478–563. Oxford Science Publications, 1995.

[132] Colin Stirling. A generalisation of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.

[133] R. Stoll. *Set Theory and Logic*. Dover, 1979.

[134] Wilson A. Sutherland. *Introduction to metric and topological spaces*. Oxford Clarendon Press, 1975.

[135] F. Swainston. *A Systems Approach to Programmable Controllers*. Newnes, 1991.

[136] P. S. Thiagarajan. Elementary net systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, volume 254 of *Lecture Notes in Computer Science*, pages 26–59, 1986.

[137] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, 1998.

[138] S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device drivers: from design to implementation. In Ramming [117].

[139] Scott Thibault. *Domain-specific Languages: Conception, Implementation and Application.* PhD thesis, L'Université de Rennes 1, 1998.

[140] Konstantinos Tourlas. Semantic analysis and design of languages for programmable logic controllers. Master's thesis, Department of Computer Science, The University of Edinburgh, 1996.

[141] Konstantinos Tourlas. An assessement of the IEC 1131-3 standard on languages for programmable controllers. In Peter Daniel, editor, *16th International Conference on Computer Safety, Reliablity and Security (SAFECOMP'97)*. Springer-Verlag, 1997.

[142] S Üsküdarlı and T. B. Dinesh. Towards a visual programming environment generator for algebraic specifications. In *Proc. 1995 IEEE Symposium Visual Languages*, September 1995.

[143] L. Walton and J. Hook. Message specification language (MSL): A domain-specific language for message translation and validation. OGI-CSE-TR, 1994.

[144] Charles Wells. Sketches: Outline with references. Unpublished, 1994. Available electronically as `http://www.cwru.edu/artsci/math/wells/pub/papers.html#sketch`.

[145] Tanya Widen and James Hook. Software design automation: Language design in the context of domain engineering. In *Tenth International Conference on Software Engineering and Knowledge Engineering*, June 1998.

[146] Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing Series. The MIT Press, 1994.

[147] Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The Rely-Guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[148] J Zhang and D Norman. Representations in distributed cognitive tasks. *Cognitive Science*, 18:87–122, 1994.

[149] M. Zhou. *Petri nets in flexible and agile automation.* Kluwer Academic Publishers, 1995.