UNIVERSITY OF SOUTHAMPTON

The Formalisation of Software Development using MASCOT

Stephen Edward Paynter

Doctor of Philosophy

MATHEMATICS

September 1993

This Thesis was submitted for examination in September 1993. It contains the modifications requested at its defence.

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF MATHEMATICAL STUDIES

MATHEMATICS

Doctor of Philosophy

THE FORMALISATION OF SOFTWARE DEVELOPMENT USING MASCOT

by Stephen Edward Paynter

Formal techniques are developed for increasing confidence of software systems developed using the MASCOT-3 design method. Rather than attempt to handle all the complexity of MASCOT-3, a subset of well-formed MASCOT designs is defined graph theoretically. Two abstract syntaxes for MASCOT subsets are defined using edge labelled neighbourhood controlled embedding graph grammars. The richest MASCOT subset given an abstract syntax consists of designs which have loops, arbitrary branching of paths on activities, but which only have single reader and single writer IDAs. This is more restrictive than the graph theoretically defined well-formed MASCOT designs. After a survey of formal semantics for concurrent languages, the abstract syntax of MASCOT is used to structure the definition of a linear interleaved semantics. The semantic model is defined using CSP. A small example is given to illustrate the safety correctness properties that may be proven for formal MASCOT designs with these semantics.

A graphical mode-based specification notation for reactive systems is developed, known as Specification Transition Systems, or STSs. STSs are a concise way of presenting large labelled transition systems (LTSs), a mode of an STS representing a class of LTS' states. STSs achieve this through the identification of sinks which model the devices which the system controls; and the definition of a dependency relation between sinks, which partitions the sinks into classes of devices which need to be receive a consistent view of an STS's mode. A formal semantics is defined for STSs in terms of allowable histories. A calculus is defined for demonstrating refinement between STSs. Also, the construction of LTSs which correspond to STSs is sketched, and the relationship between STSs and process-based designs is investigated.

Table of Contents

Acknowledgements

I would like to thank Prof. Bernard Carré and Dr. Ray D'Inverno, my supervisors, for many helpful discussions, good advice, and probing questions. I also owe a debt of thanks to Mr. Bob Born for many stimulating conversations over the years. Mr. William Marsh's advice and comments have also been much appreciated, I especially want to acknowledge that the idea of using graph grammars was due to him. I would also like to thank Prof. Bruce Batchelor for much encouragement and advice.

I would like to thank British Aerospace Defence Ltd., Dynamics Division, for employing me during this research. In particular, I would like to thank Mr. Peter Marshall, Mrs. Christine Thomas, Mr. George Woodward, and Dr. Hugo Simpson for their support. I also enjoyed interesting discussions with Mr. Mike Harding, and Mr. Dave Nuttall.

Lastly, I want to acknowledge an immense debt to Helen, my wife. Without her encouragement the research would not have been started, this thesis would have taken much longer, and would have been in poorer English.

LORD, ...
Everything we have accomplished
You have done for us.
Isaiah, Chapter 26, verse 12.

Proof is the bottom line for everyone
Paul Simon,
Proof, The Rhythm of the Saints

Dedication

To Helen, for everything.

Symbols

Numeric and arithmetic symbols:

-1, 0, 1, 2, ...	numerals representing the integers
N	the set of all positive integers
-	natural number subtraction
+	natural number addition

Logical and Meta-logical symbols:

$\neg$	not
$\wedge$	and
$\vee$	or
$\Rightarrow$	implies
$\forall$	for all, universal quantification
$\exists$	there exists, existential quantification
•	such that, a separator between quantifiers, and quantified expression.
$H \vdash_F E$	A sequent: E is provable from hypothesis H, in formal system F.
$M \models_F E$	E is "true" when interpreted in model M of formal system F.

Set theoretic symbols:

$a \in X$	set membership: a is a member of set X.
$A \cup B$	set union: the union of sets A and B.
$A \cap B$	set intersection: the intersection of sets A and B.
$A \subseteq B$	subset: set A is a subset of set B.
$A \subset B$	proper subset: set A is a proper subset of set B.
$\{ t \mid P(t) \}$	set comprehension: the set contains all t's such that P(t) holds true, where P is a first-order predicate expression over t.
$\wp(X)$	the power set of X, where X is a set.
X^*	the set of all possible sequences of the elements of the set X.

Sequence theoretic symbols:

A-seq	the type: sequence of elements from set A.
<>	the null sequence.
a^L	the infix concatenation operator. ^ : A-seq × A-seq → A-seq
head(L)	a function which returns the start of a sequence. head : A-seq → A
tail(L)	a partial function which removes the head of a sequence. tail : A-seq → A-seq
len(L)	a function which returns the length of a sequence. len: A-Seq → N

CSP symbols:

STOP	the process which does nothing
$(c \rightarrow P)$	the process that does c, and then behaves like process P.
$(c \rightarrow P) \mid (d \rightarrow Q)$	the process either does c, and then P, or d and then Q.
$(x{:}B \rightarrow P(x))$	the process may do any action from the set B, and then behave like P(x).
(P/s)	the process that behaves like P does after it has performed s.
(P\A)	the set of actions A is hidden from the alphabet of P.
$(P \parallel Q)$	the processes P and Q executing concurrently, synchronising on shared events, that is, events in both processes alphabets.

$(P \sqcap Q)$	the internal choice between processes P and Q.
$(P \gg Q)$	the parallel processes P and Q are connected by a pipe. This is a derived operator, and is defined as P and Q in parallel, with their shared actions over a common channel hidden.
$\mu X:A.F(X)$	the recursive process with alphabet A, defined by F.
F^n	n iterative applications of the process F.
f(P)	f is a re-labelling function which changes the names of actions in the alphabet of P.

CSP semantic operators:

<>	the empty trace.
s^t	the concatenation of two traces.
t_0	the first element of the trace, t.
t'	the tail of the trace t.
A*	the set of all possible traces whose elements are drawn from the set A.
f*(t)	The function which applies the function f to each element in the trace t.
$t \lceil A$	the restriction of trace t to elements in A.
αP	the alphabet of process P.
$s \leq t$	s is a, possibly empty, prefix of t.
$s \leq^n t$	s is a, possibly empty, prefix of t, and the length of s is no more than n shorter than t.
#t	the length of the trace t, a natural number.

Other symbols:

$A \sqsubseteq B$	A is refined by B.
$\triangleq$	is defined as.
$fn : A \rightarrow B$	"fn" is a function which takes values of type A, and returns results of type B.

STS symbols:

$m \rightarrow^a m'$	for a given STS, $<M, M_0, A, T, ...>$, $(m, a, m') \in T$.

Chapter 1: Introduction

This thesis is intended as a contribution to formal methods for the construction of embedded software systems. In particular, formal techniques are presented for increasing confidence in software systems developed using the MASCOT design method, [JIM87]. MASCOT enables concurrent and distributed software designs to be employed. Time and the issues of temporal requirements, specification and correctness are not treated quantitatively in this thesis. This is primarily because, although MASCOT is often used to design real-time systems, it contains no notation for recording timing behaviour or requirements.

In this thesis a subset of MASCOT designs is considered. The abstract syntax of these designs is defined using a graph grammar. The abstract syntax is used to structure the definition of a formal semantics of MASCOT. The semantic model used is Hoare Traces, [Hoa85]. A mode-based specification notation, known as Specification Transition Systems, or STSs, is defined, and is given a formal semantics. A refinement calculus for STSs is defined.

This introduction gives the motivation for the work by discussing the problems of safety-critical software, and the roles of design methods and formal methods in software development. It also includes an overview of the content and structure of the thesis.

1.1. Safety-Critical Software

An important and growing class of products have microprocessors as components. The microprocessors and software of such products are known as *embedded systems*. The flexible nature of computer programs, and the ease of encoding complicated algorithms in software, means that embedded systems are often responsible for controlling the behaviour of a product. Some products, however, such as fly-by-wire aeroplanes, agile missiles, nuclear power stations, and railway signal-controllers, may exhibit dangerous, life-threatening behaviour if they fail to function as intended. Often it is the task of an embedded system to ensure that correct, and non-hazardous, behaviour is exhibited. The software of such embedded systems may be termed *safety-critical software*, often abbreviated to the acronym: "SCS".

Clearly it is important that safety-critical embedded systems function correctly. A significant factor in this will be the correct functioning of the software. Fortunately,

software is not subject to deterioration. Any undesirable behaviour a program exhibits reflects an error that has been present since it was written. It is hence worthwhile to investigate how a program will behave before it is used in a potentially hazardous environment.

One common way of attempting to determine a program's behaviour is to *test* it. This will involve examining the outputs the program produces for a range of inputs. The very large number of possible input combinations to any but the most trivial of programs makes it infeasible to test a program exhaustively. This is especially true of embedded software, which will often exhibit behaviour dependent upon the precise temporal ordering of the inputs' arrival. Testing, therefore, can only provide evidence of the software's behaviour for a limited number of scenarios. It cannot say how the software will behave in other situations. This is especially significant for computer systems, where small input variations can produce widely different output behaviours, due to their discontinuous nature.

These limitations of testing have led many to assert that testing is incapable of providing enough assurance in the behaviour of a program to be acceptable for safety-critical software. It has been suggested that testing should be supplemented by formal reasoning about the behaviour of the algorithms used in the software. The goal is to prove mathematically that the software is unable to exhibit behaviour which will drive the product in a dangerous way; and that it will not fail to drive the product in a safe way.

However, reasoning about embedded systems is a hard task. A minimum pre-requisite is a formal model of the system; a model from which it is possible to deduce behaviour. Developing such a model is complicated by the fact that embedded systems are usually reactive[1]; the software is often structured into concurrent programs which interact; and it may well be the case that the hazardous behaviours are time-dependent in some way.

Reasoning is simplified if the software has a sensible structure and if there is a formal statement of what it is intended to do. It is the role of design methods, introduced in the next section, to promote the sensible structuring of software into well-partitioned units. A formal specification is helpful as it is easier to reason about behaviour using more abstract expressions of the software's functionality, and to prove that a program implements a specification, than it is to reason directly about the behaviour of a program. Formal methods essentially comprise a specification language and a development method; both with

[1] A discussion of reactive systems can be found in Appendix Two.

mathematical underpinnings. Thus the use of formal methods has been advocated in the development of safety-critical software, [BaM92].

As well as functioning correctly, software for safety-critical systems also ought to exhibit other properties, such as safe behaviour in the presence of hardware failure. However, such considerations are beyond the scope of this thesis.

This thesis is presented in the context of safety-critical systems because safety and safety standards, for example [MoD91b], are the most significant forces in motivating the development and adoption of formal methods, although there are other influences, including: security; quality; and commercial considerations. Formal methods are discussed in Section 1.3.

1.2. The Role of Design Methods

A software design method often has at least some of the following features:

* Guidelines (rules of thumb) for partitioning the software into manageable units
* A notation (usually graphical) for recording the structure of the software design
* Rules or guidelines on how the design should be documented.
* Rules for checking the consistency of a design
* Rules or guidelines on how to develop software to fit the design structure
* Rules or guidelines on how to test the resulting software

Examples of design methods which have been advocated include: Structured Design, [SMC74], JSP (Jackson Structured Programming), [KiP85]; JSD (Jackson System Design), [Cam86]; Yourdon Structured Method, [Woo88]; SSADM (Structured Systems Analysis and Design Method), [Ash88]; OOD (Object-Oriented Design), [Boo86] and [Boo91]; HOOD (Hierarchical Object-Oriented Design), [Rob88]; and MASCOT-3 (Modular Approach to Software Construction, Operation and Test), [JIM87]. [Ber81] is a survey of the earlier methods, and [HOH91a] is a more recent comparison of design methods for real-time systems.

Design methods encourage the use of a consistent policy in partitioning software, and, to some extent, propose good practice in software structuring. A good design notation provides a structured way of thinking about systems at an abstract level. These methods usually

have no formal foundation, yet they have proved popular in developing large software systems.

It should be noted that design methods usually do not enable the functionality of the system to be expressed. They are limited to defining the different software units of the design, and the interfaces between the units. The functionality that design notations can define is usually incomplete, and is expressed in terms of some informal intuition of what a graphical symbol represents.

Clearly, design methods cannot, by themselves, be used to generate the level of confidence required for safety-critical software. Nevertheless, it is claimed that the structuring which good design methods encourage means that they have a role in the development of large safety-critical programs. It is believed that the structuring guidelines embodied in design methods actually capture important experience[2], and that designs which follow such guidelines are likely to be easier to reason about than those which do not.

1.3. The Role of Formal Methods

Formal methods are approaches that use logic and mathematics in developing computer programs, with the aim of increasing knowledge of the behaviour of a program so developed. Formal methods essentially consist of a specification language and a development method. The specification language usually has a formal syntax and a formal semantics. That is, a well-formed specification is taken to denote a (possibly empty[3]) class of mathematical objects. There may also be a proof theory for the specification language; this enables properties of the specification to be proven. Clearly such proof theories must be sound with respect to the language's semantics[4].

Development methods which are associated with formal methods are not prescriptive, that is, they do not define how to construct a program, [Coo92]. Instead they provide the framework in which programs can be formally related to specifications. A program which can be formally related to its specification are said to be a "refinement" or "reification" of it.

---

[2] To embody such guidelines a design method need not be prescriptive concerning how to develop designs, nor do the guidelines need to be explicit. Design guidelines may be transmitted in the philosophy of the method, in the case studies in the accompanying literature, or be a result of using the design notation.

[3] A well-formed, syntactically correct, specification may still be self-contradictory, and so denote nothing.

[4] A basic introduction to formal systems, including the definition of basic terms such as "soundness" can be found in Appendix One.

Formal methods contain some formalised definition of what it means for one expression to be a refinement of another. Expressions related by refinement can be viewed as forming a hierarchy of levels. Demonstrating refinement ensures that the functionality of one level can be exhibited by the next. Demonstrating that two levels are related by refinement usually involves discharging proof obligations. The fact that refinement is transitive enables the proof of refinement to be broken down into a number of steps, each demonstrating refinement between adjacent levels of abstraction.

Formal development methods are still a research area, especially development methods for concurrent systems. However, three broad approaches to demonstrating refinement have been proposed, [HaJ89]: the *constructive-type-theoretic* approach; the *derivation* approach; and the *design-and-verify* approach. In the type-theoretic approach, a program is extracted from a constructive proof of the existence of an implementation of the specification. Refinement is established by the validity of the proof and extraction rules. Examples of this approach include: [Mar82]; [BCM89]; and [Hen89]. In the derivation approach, the specification is manipulated into a design using correctness preserving transformations. Refinement is guaranteed by the soundness of the transformations. There are often proof obligations to discharge to demonstrate the applicability of a given transformation. Examples of this approach include: [MRG88]; [BMP89]; [Mor90]; and [Par90]. In the design-and-verify approach, each subsequent level is posited, and then proof obligations are discharged to demonstrate its validity, thus establishing the refinement relation. A well known example of this approach is VDM (Vienna Development Method), [Jon90]. It is the design-and-verify approach which is advocated in this thesis as it includes the posit phase where the traditional skills of the designer can be utilised. It is contended that this integrates formal methods most naturally with design methods. However, other views have been expressed, for example, [Fei93], which advocates integration within a derivation approach.

Formal methods have a number of significant advantages over traditional methods of software development. A formal specification introduces clarity, precision, and, used correctly, abstraction early into the software development process. The fact that it has a precise semantics may enable the presence or absence of certain system properties to be proven formally from the specification. A formal development ensures that close reasoning is carried out to demonstrate the correctness of the software with respect to the specification; the systematic nature of the reasoning encouraged requires the behaviour of

the software to be considered much more exhaustively than is common in traditional development methods.

The advantage of formal reasoning over other kinds of reasoning is that it is a machine-checkable symbol manipulation process. The process is unaffected by rhetoric and any intuitions about the intended properties of the formal symbols being reasoned about. The only sources of error are in the underlying formal system itself, i.e. the axioms and inference rules, and the misinterpretation of what has actually been proven. Formal arguments force underlying assumptions to be made explicit[5].

However there have been a number of serious objections raised to formal methods and formal verification; some of the most prominent of which are De Millo et al's arguments, [DLP79], and Fetzer's arguments, [Fet88]. The first of these may be summarised as arguing that proof is a social process involving peer review, but that the sheer size of proofs in formal methods prevents them from being subject to that process. De Millo et al conclude that such proofs cannot increase confidence in the quality of the software. A weakness in this argument is the assumption that machine checked proofs need to be validated by a social process. It can be argued that the peer review for formal proofs is the acceptance of the underlying formal system, and the confidence in the tool which checks or constructs the proof. Hence, the social-review problem is only applicable to the proofs of the soundness of the proof tools. Furthermore, tool confidence can be generated through testing and wide use. This is not to argue that such tools and formal systems already exist, but only that the problem is solvable.

Fetzer's argument is more interesting. It is that programs are causal entities that control computers, while verification proofs can only reason about non-causal entities, for example, the algorithms which the programs encode. Fetzer concludes that formal reasoning cannot guarantee the correctness of programs. This is, of course, correct, yet the distinction only manifests itself when the model of the algorithm and computer is different from the actual language and computer. As pointed out in [BSY89], in principle it ought to be possible to reduce this gap to the difference between the boolean "and" operator, and the "and" operator encoded in the computer's silicon. At present however this gap is significant. The semantic model of programming languages used in proofs of the correctness of algorithms are usually given axiomatically using Hoare logic, [Hoa69], or Dijkstra's weakest

[5] This explains the comment in [Lak76], that "the virtue of a logical proof is not that it compels belief but that it suggests doubts", on page 48, attributed to H.G. Forder.

preconditions, [Dij75], and these are significantly removed from assembly code which the high-level language will be compiled into. Furthermore, the differences between the functional model of the processor implicit in such models, and the processor's logical design in silicon are also significant. This gap between model and embedded system is even larger when models of concurrent software are considered, as such models usually abstract away from details of scheduling and distribution.

Formal methods suffer from significant fundamental limitations, as well as numerous limitations associated with their current development. Perhaps the most significant of the fundamental limitations is that although the software may be proven correct with respect to the specification, the specification itself cannot be verified, only validated, [BaM92]. Two main schools of thought have developed concerning how it is best to validate a formal specification. One, argued in [HaJ89], recommends the use of the most abstract and powerful mathematical concepts suitable to express the specification of the system, thus enabling the easiest comprehension by the reader, and making it easiest to reason about the system's behaviour. The other, argued in [Fuc92], recommends the use of executable subsets of mathematics in specification, so that a specification can be tested by executing it, and hence the properties of the system specified can be investigated interactively. Fuchs, in [Fuc92], argues that a modern logic programming language can be used to present a specification at the same level of abstraction as a non-executable specification. Fuchs illustrates this argument using the examples that were used in [HaJ89].

However, neither of these solutions to specification validation can generate complete confidence that the specification actually specifies the desired system. They both share a limitation of testing, naming only certain properties will be proven, and limited scenarios animated. Furthermore, the properties of the desired system may be ill defined, and poorly articulated. Indeed, it is not uncommon for different people (users and customers) to have contradictory ideas of the properties that the system should exhibit.

Other fundamental limitations of formal methods include: ambiguity in formal specifications, irrespective of the precision of their semantics, as they have to be interpreted by people; and the inability to express or prove certain properties in a given formal system, [BaM92]. For these reasons, amongst others, formal methods should never completely supplant the testing of the final software and product.

It has been argued, [BaM92], that formal specification languages are currently too arcane. There have been three basic responses to this: one is to deny it, and contend that

symbols quickly become invisible as their meaning is apprehended, [Hoa85]; another is to develop a graphical representation of the formal language, for example, [Arm92]; and yet another is to start from a graphical notation, and to develop a formal semantics for it. While there is some merit in the first position, it is the last that is pursued in this thesis. The main reason for this was the desire to develop a formal development method which was as close as possible to current, industrial, practices. For example, currently within BAe Dynamics Ltd.[6], mode-based graphical specifications are sometimes used, usually with only informally defined semantics. The specification language proposed in this thesis is a mode-based graphical notation with a formal semantics. It is hoped that this position will ease the dissemination within industry of the formal methods proposed in this thesis.

To summarise; I have argued that the use of formal methods can increase confidence in a software system's behaviour, and hence that formal methods have a role in developing safety-critical software. I also believe that the advantages of well-structured software means that design methods also have a role in developing safety-critical software. Hence there is a need to produce formal semantics for design notations, so that designs produced using these methods can be reasoned about formally. I also believe that graphically based specification notations make formal notations more usable.

This basic position motivates the work in this thesis, which is described in detail in the next section. The design method which is formalised is MASCOT, [JIM87], and the graphical specification language is a mode-based notation suitable for describing reactive systems.

1.4. Introduction to the Thesis

This thesis has seven chapters. After this introduction, the second chapter briefly introduces the MASCOT design method and some of the extensions that have been proposed during the DORIS[7] applied research project at BAe Dynamics, [Sim91]. MASCOT was chosen over the other design methods for two main reasons. Firstly, MASCOT is widely used in the UK defence industry for producing software, being the Ministry of Defence's

[6]The author is an employee of British Aerospace Dynamics Ltd., (referred to as BAe Dynamics in the rest of the thesis).

[7]Data Oriented Requirements Implementation Scheme

preferred design method, [MoD85], [MoD91a][8], and it has been, and is being, used to develop safety-critical software. Secondly, MASCOT is considered by many to be one of the better design methods, especially for concurrent embedded systems. [HOH91a] is a fair comparison of MASCOT with other major design methods, including MOON, an object-oriented extension of MASCOT, [HOH91b]. Although MASCOT has been used in the development of SCS, there are no widely used formal semantics for the method; in spite of one having been defined in [BJP87]. The main MASCOT extension from the DORIS work which is considered in this thesis is the Signal IDA[9]. The second chapter defines a simplified MASCOT, and gives a graph theoretic characterisation of well-formed MASCOT designs.

A simplified MASCOT is considered in this thesis, rather than the whole MASCOT-3 method, as the aim was to be able to develop SCS in a MASCOT-like way, rather than to formalise MASCOT per se. The position has been taken that it is acceptable to exclude the use of certain aspects of MASCOT on safety-critical projects when those aspects are difficult to reason about. This is analogous to the use of "safe" programming language subsets for SCS, [Car90] and [CGW91].

The third chapter discusses the ways that a design method may be formalised; and it defines the abstract syntax of two simplified MASCOT-like design notations. The abstract syntax is defined for the MASCOT graphical language, and so graph grammars are used. In particular, directed edge-labelled neighbourhood controlled embedding (deNCE) graph grammars are used. These are a particular kind of Node Labelled Controlled (NLC) graph grammar, and are properly introduced in the third chapter. The first syntax, known as Simple Linear MASCOT (SLM) is an especially simple class of designs. The second, Branching Looping MASCOT (BLM) is more complicated, but it is still simpler than the class of well-formed MASCOT designs defined in chapter two. The BLM abstract syntax is used in chapter five to structure the definition of a denotational semantics for MASCOT.

The fourth chapter reviews various formal models that have been proposed for concurrent systems with respect to their appropriateness for being a semantic foundation

[8] The Navy's commitment to MASCOT is strong, as revealed by the following statement from [MoD91a]: "For real-time multi-tasking systems the design methodology and philosophy of MASCOT, (DEF STAN 00-17) (or for use with Ada, MASCOT 3), is to be used unless the Contractor can demonstrate that an alternative approach is better suited for the application."

[9] IDAs (Inter-communication Data Areas) and other MASCOT acronyms and concepts are introduced in Chapter 2, and in the standard MASCOT literature such as [JIM87] or [Sim86].

for MASCOT designs. This chapter concludes that Hoare Traces, [Hoa85], can be used as a suitable semantic model for MASCOT designs.

Chapter five uses this conclusion and defines a denotational semantics for SLM and BLM MASCOT designs in terms of Hoare Traces, by being defined in terms of CSP[10], [Hoa85]. The utility of this semantic model is illustrated with the proof of correctness of a simple MASCOT design with respect to a predicate-over-traces specification using the normal CSP SAT logic, [Hoa85].

Chapter six defines a mode-based notation called Specification Transition Systems or STSs. The modes of an STS define a class of labelled transition system (LTS) states. However, STSs are not more expressive than LTSs, and this is demonstrated by defining the construction of an equivalent LTS from an STS. STSs contain a notation to identify the devices which the system is to control. STSs also contain a notation to define which devices need to be given a consistent view of the system's mode. A graphical presentation notation and a refinement calculus for STSs is defined. The mapping between MASCOT designs and STSs is discussed, but not formalised.

The seventh chapter summarises the thesis, evaluates the work, and identifies areas for further work. There are five appendices to the thesis. The first introduces formal (logical) systems; the second introduces reactive systems; the third gives the formal definitions of the models of concurrency which are mentioned in the thesis; the fourth contains proofs of the soundness of the STS refinement rules; the fifth contains an outline of a ten rule deNCE grammar for BLM designs; and the sixth contains the formal version of the BLM_GG grammar given graphically in chapter three. The first three appendices are included in an attempt to make the thesis self-contained and accessible to a wider audience.

[10]Communicating Sequential Processes

Chapter 2: Introduction to MASCOT Designs

This chapter introduces the MASCOT design method. "MASCOT" is an acronym for "Modular Approach to Software Construction, Operation and Test". MASCOT has been advocated for the design of large, concurrent or distributed, real-time, embedded software systems. MASCOT is widely used in the United Kingdom defence industry, being a preferred design method of the Ministry of Defence, [MoD85] and [MoD91a]. MASCOT-3 is the latest version of the MASCOT design method. [JIM87] is the official definition of MASCOT-3, other introductions to MASCOT include: [SiJ79], [Bat86], and [Sim86].

The designs considered in this chapter are a subset of the valid MASCOT-3 designs; they make use of some of the simplifications and extensions to MASCOT that have been proposed during the DORIS research project, [Sim91]; [Tho91]; and [Sim93]. There are two main motivations for introducing a simplified MASCOT. The first is that there are elements of MASCOT-3 designs which, while useful for presenting large scale designs, actually add no new operational entities. As the intention is to present a formal semantics of MASCOT designs so that their behaviour can be reasoned about, there is a case for only considering designs consisting of entities which affect the behaviour of the design. This case is strengthened by the realisation that the removal of these non-operational entities can result in a dramatically simpler characterisation of MASCOT.

A second motivation for considering a simplified MASCOT is the desire to remove from consideration certain MASCOT entities for which it would be hard to provide a formal semantics. For example, the behaviour of general IDAs is under-defined in MASCOT-3. It is anticipated that, even if a formal semantics were defined for them, they would not have a powerful enough theory associated with them to enable anything interesting to be deduced about the behaviour of designs which contained them.

2.1. Activities and IDAs

There are two basic kinds of component in a MASCOT design: activities and intercommunication data areas (IDAs). Activities are the active processes (or tasks, or agents) of the design, and IDAs are, conceptually, the passive storage areas through which activities communicate. IDAs are only "conceptually passive" because they may be

implemented by active processes in a given implementation of a design[11]. This is the case, for example, with the proposed mapping between MASCOT designs and Occam, [INM88], in [Kno90]. However, at the MASCOT design level of abstraction, IDAs appear passive. They store the data in transit between activities, and do not engage in independent processing of their own. Activities may only communicate via IDAs. MASCOT designs are static networks of activities and IDAs.

There are three basic kinds of IDA, known as: *pools*; *channels*; and *signals*. They are distinguished by the protocols they impose upon the communication of data which passes through them. Pools, for example, are like shared variables. Their value is unaffected by activities reading from them, and activities which write to them overwrite their previous value. Pools are said to have a destructive writing and a non-destructive reading protocol. In contrast, channels have a buffer associated with them, and data is consumed from the buffer by a reading activity, and is added to the buffer by a writing activity. Reading activities must wait when a channel's buffer is empty, and writing activities must wait when the buffer is full[12]. Channels are said to have a non-destructive writing and a destructive reading protocol. Signals, like channels, also have buffers associated with them, however signals have a destructive writing and reading protocol. Signals impose a circular overwriting protocol on the buffer for writing activities. Signal IDAs are an extension proposed in [Sim91], they are not distinguished in MASCOT-3.

The above information about the communication protocols of the different IDAs is captured in the table in Figure 2.1, which was taken from [Sim93]. The "Constant" IDA mentioned in the table is included in the DORIS work for completeness. Constant IDAs indicate the use of fixed configuration data by an activity. They will not be considered further in this thesis.

[11]The MASCOT-3 Handbook, [JIM87], does not draw this distinction, and simply asserts that, "An IDA is a passive element".

[12]The buffers associated with MASCOT channels are intended to be implemented and so are always of finite length.

	Destructive Reading	Non-Destructive Reading
Destructive Writing	SIGNAL	POOL
Non-Destructive Writing	CHANNEL	CONSTANT

Figure 2.1: IDA Communication Protocols

The data propagation protocols of IDAs have a natural impact on the synchronisation of activities that communicate via them. This can be characterised by whether or not the reading or writing activities can be forced to wait for data from the IDA. This characterisation of the kinds of IDA is presented in Figure 2.2, which was taken from [Sim93].

	Reader can be held up	Reader cannot be held up
Writer cannot be held up	SIGNAL	POOL
Writer can be held up	CHANNEL	CONSTANT

Figure 2.2: IDA Synchronisation Protocols

Pools are suitable for storing information that needs to be shared by more than one activity, and for de-coupling activities which produce and consume data at different and, particularly, varying frequencies. Signals are suitable for communicating messages between activities where it is more important for the consuming activity to process the latest message than for it diligently to handle each message in turn. This is common is real-time systems. Channels are suitable for communicating messages where no message may be lost, even if this forces one or more of the activities to wait from time to time.

It is the richness in communication and synchronisation mechanisms which are provided by IDAs that is the essential advantage of MASCOT designs over other real-time design approaches. For example, consideration of a channel with a zero-place buffer reveals that the rendezvous mechanism of Ada, [LRM83], is a special case of the MASCOT channel IDA.

Pools can be implemented in such a way that truly asynchronous communication is possible between activities which execute on distributed processors. In particular, the four-slot mechanism, [Sim90b], [Sim92] and [Ben92], can be used to ensure the coherence of data communicated via a pool without introducing critical regions, and hence synchronisation[13].

MASCOT-3 designs may contain general IDAs, that is, IDAs without an explicit protocol associated with them. There are few things that can be deduced about the behaviour of a design which contains such an IDA without knowing more about its functionality. A model may be found for a design with a specific example of such an IDA, but it is hard to find a powerful enough semantics for the general case. For example, a particular design may use a general IDA to represent a complicated intercommunication data area, such as a database. Usable bespoke models may be able to be defined for such specific examples. Nevertheless, general IDAs are not considered further in this thesis.

MASCOT-3 also supports the definition of the interfaces between activities and IDAs. These interfaces, known as "access interfaces", may include data types, variables, and procedures and functions. An access interface is defined by a "window" at the IDA, and it's signature (the signature of the types, variables, procedures, and functions) must be common to a "port" associated with every activity connected to that window.

The MASCOT designs considered in this thesis are simplified in this aspect. Each IDA supports two implicit windows, one for a reading activity and one for a writing activity. The access interfaces for these windows support a single procedure or function, namely a writing procedure or a reading function. The semantics of these operations are such that the data value being communicated is passed from the activity to the IDA or vice versa without being modified, and the IDA is updated appropriately according to the communication protocol it imposes. These simple default access interfaces for IDAs mean that windows and ports can

[13]The asynchronism possible in an implementation of a MASCOT design is therefore fundamentally different from the asynchronous process communication discussed in [JHH89], which is with communication through buffers of unbounded capacity.

be dropped from IDAs and activities. Any activity reading or writing to an IDA will be automatically assumed to be using the implicit access interface. Ports, windows, and access interfaces have an important role in making the consistency of a MASCOT-3 design automatically checkable, as well as flexible. However, for formal reasoning about MASCOT designs, such complexity is an undesirable overhead.

2.2. Interfacing with the Environment

A MASCOT-3 design interacts with its environment via special network components known as *servers*. A server is intended to collect together the software required for handling a particular device in the system's environment. A server, therefore, includes any interface software (i.e. memory mapped variables; and the procedures and functions which manipulate them) and any interrupt handlers associated with the device. A server (with interrupt handler) may hence be viewed as a partially active, as well as a passive, component. Similarly to a MASCOT-3 IDA, a server may provide a number of different access interfaces to the underlying device via different windows.

The designs considered in this thesis take a simpler view than MASCOT-3 of design environment interfacing. Rather than collecting together in a "server" a number of hardware access routines, each hardware access will be handled separately. Furthermore, rather than allowing arbitrary access interfaces, simple "read" and "write" routines will be assumed, which transport, but do not manipulate, the data being passed between the activities and the hardware. Another simplifying assumption adopted is that the synchronisation protocol for an activity communicating with the hardware will be one of the ones associated with the three kinds of IDA described above.

The consequence of these simplifications is that a MASCOT design may be considered to be a network of activities and IDAs components, and each path through the network will be bounded by interface IDAs. Interface IDAs would be connected to either consuming activities or producing activities within the design, and to (implicit) devices in the environment, which either produced or consumed the data on the other side. The advantage of this view is that the MASCOT designs considered in this thesis consist of only two kinds of components: activities and IDAs, and a third kind for interfacing is not required. This keeps the ontological commitments of MASCOT to a minimum.

The equivalent of interrupt handlers could be modelled in this approach by providing a dedicated activity connected to the appropriate hardware device by a signal interface IDA, with zero length buffer.

Servers have also been dropped from MASCOT in the DORIS work, and DORIS networks are also bounded by IDAs, [Sim93].

2.3. MASCOT Networks

A MASCOT design is a network of activity and IDA components. It can be viewed as a node labelled graph, where the connections of the network (the arcs of the graph) indicate the paths by which data may be communicated between these components. Thus, if an activity writes data to a pool, there is be a connection between the activity component and the pool component. In the MASCOT designs considered in this thesis, as the only access interfaces supported are "reads" and "writes", a network may be considered to be a directed graph, where the arcs indicate the direction of the flow of data between the components.

The fact that activities may only communicate via IDAs means that activities and IDAs must label alternating nodes of a MASCOT network on each path through the network.

There are other desirable constraints to be placed upon how components may be connected in MASCOT designs. In particular, pools should only be connected to a single producer activity, and channels and signals should only be connected to a single consumer activity. Also no interface IDA may be written to by more than one activity. Output interface IDAs are treated differently than other IDAs to keep them simpler as they define the hardware / software boundary. It prevents the problem of two different activities sending conflicting values to the same place, and means that an implementation of such an IDA does not need to implement mutual exclusion, or other protocols to ensure data coherence[14].

2.3.1. Terminology

A *directed graph* is a pair, G = (N, A), where N is a finite set of elements known as *nodes*, and A is a subset of the cartesian product N × N, whose elements are known as

[14] "Shared resources" can still occur in these simplified MASCOT designs, providing the MASCOT design explicitly provides an activity to arbitrate access to such resources.

arcs. For an arc, $a \in A$, where $a = (x, y)$, x and y are the *endpoints* of a, x is known as the *initial endpoint* of a, and y is the *terminal endpoint* of a. The initial and terminal endpoints of an arc a may also denoted by a_i and a_t, respectively. Two nodes are *adjacent* if they are joined by an arc. Two arcs are *adjacent* if they share a common endpoint. A node x is said to be an *immediate predecessor* of a node y iff $(x, y) \in A$. Similarly, y is said to be an *immediate successor* of x iff $(x, y) \in A$. The set of all immediate successors to a node n will be denoted by $\Upsilon^+(n)$, and the set of all immediate predecessors to a node n will be denoted by $\Upsilon^-(n)$. The set of all *neighbours* to a node n is $\Upsilon^+(n) \cup \Upsilon^-(n)$. A *subgraph*, H, of graph, G, is $H = (Y, A_Y)$, where $Y \subseteq N$, and $A_Y = A \cap Y \times Y$. (These definitions are taken from [Car79].)

2.3.2. Formal Characterisation of MASCOT Designs

The MASCOT designs which are considered in this thesis may be characterised as a directed graph,

MASCOT_DESIGNS ::Nodes : COMPONENTS
Arcs : COMPONENTS × COMPONENTS

Four helpful functions are:

Node_Type : COMPONENTS → { IDA, Activity },

which returns the kind of the component. Node_Type is a total function.

IDA_Type : COMPONENTS → { Pool, Channel, Signal },

which returns the type of IDA. The precondition of IDA_Type is that the value returned by applying the Node_Type function to the parameter is "IDA".

Predecessors : COMPONENTS → COMPONENTS-set

which returns the set of all predecessors to the node, that is, all immediate predecessors, and their predecessors. When the set returned is empty the node is said to be an input node.

Successors : COMPONENTS → COMPONENTS-set

which returns the set of successors to the node, that is, all immediate successors, and their successors. When the set returned is empty the node is said to be an output node.

The following well-formedness rules apply to valid MASCOT networks:

i) dom Arcs $\cup$ ran Arcs = Nodes

ii) $\forall$ a $\in$ Arcs • Node_Type(a_i) $\neq$ Node_Type(a_t)

iii) $\forall$ n $\in$ Nodes • Predecessors(n) = $\emptyset$ $\Rightarrow$ Node_Type(n) = IDA

iv) $\forall$ n $\in$ Nodes • Successors(n) = $\emptyset$ $\Rightarrow$ Node_Type(n) = IDA

v) $\forall$ n $\in$ Nodes • $\exists$ n' $\in$ Nodes •
n' $\in$ Predecessors(n) $\wedge$ Predecessors(n') = $\emptyset$ $\Rightarrow$ Node_Type(n') = IDA

vi) $\forall$ n $\in$ Nodes • $\exists$ n' $\in$ Nodes •
n' $\in$ Successors(n) $\wedge$ Successors(n') = $\emptyset$ $\Rightarrow$ Node_Type(n') = IDA

vii)$\forall$ a, a' $\in$ Arcs •
$a'_t = a_t \wedge$ Node_Type(a_t) = IDA $\wedge$
(IDA_Type(a_t) = Pool $\vee$ Successors(a_t) = $\emptyset$) $\Rightarrow$ a' = a

viii) $\forall$ a, a' $\in$ Arcs • $a'_i = a_i \wedge$ Node_Type(a_i) = IDA $\wedge$ IDA_Type(a_i) $\neq$ Pool $\Rightarrow$ a' = a

2.3.3. Well-Formedness Rules

Informally the well-formedness rules state:

1. Each component is connected to another component.

2. Every arc must connect an activity at one end, and to an IDA the other end. This is the standard MASCOT restriction on direct activity to activity communication. It is easier to state for these networks, where the complexity of MASCOT-3 composite IDAs (see below) has been avoided.

3. All start nodes are IDAs.

4. All end nodes are IDAs.

5. Every node is connected by a path to an input node. No (cyclic) part of a MASCOT design is not connected to an input device.

6. Every node is connected by a path to an output node. No (cyclic) part of a MASCOT design is not connected to an output device.

7. An IDA may only be connected to more than one writing activity if it does not have a pool dynamic protocol, and it is not an output interface IDA. The motivation for this restriction is to keep the semantics of pools as simple as possible.

8. An IDA may only be connected to more than one reading activity if it has a pool dynamic protocol. Again the motivation for this restriction is to keep the semantics of IDAs as simple as possible.

The above well-formedness rules imply that MASCOT designs have certain desirable properties. For example:

A. For every input to the design, there is be a path through the network by which it may influence an output[15]. This does not guarantee that every input will be used to generate an output, only that the design must not be such that an input is prevented from influencing at least one output.

B. There is no need for a network to be connected, in a graph theoretic sense. That is, a design need not consist of only one connected component. It is perfectly valid for a MASCOT design to consist of two or more totally independent networks for processing inputs and producing outputs.

The characterisation of MASCOT designs above has been given to explain, from a MASCOT perspective, how activities and IDAs may be interconnected. However, the designs which will actually be considered in this thesis are defined by graph grammars rather than as a graph. The reasons for this are that graph grammars are needed for the structuring the formal semantics of MASCOT designs. Unfortunately, a graph grammar will not be presented which can generate the same set of graphs as defined by the well-formedness rules above. While there is a need for such a grammar, it would be significantly larger than the small grammars presented in this thesis.

2.4. MASCOT-3 Hierarchical Structuring Mechanisms

The MASCOT-3 design notation has been developed as a design notation for recording large scale software designs, and hence has a number of structuring mechanisms to enable large MASCOT networks to be presented economically. Servers, access interfaces, windows

[15] Clearly, an input which cannot influence the output behaviour of the system is an input which is not needed.

and ports have already been described. Other MASCOT-3 mechanisms are described briefly in this section for completeness, but they are not used in the rest of the thesis.

As a guide to the relationship between the terms used in the following discussion, a map of the various MASCOT-3 module terms and their classification is given in Figure 2.3, reproduced from [JIM87].

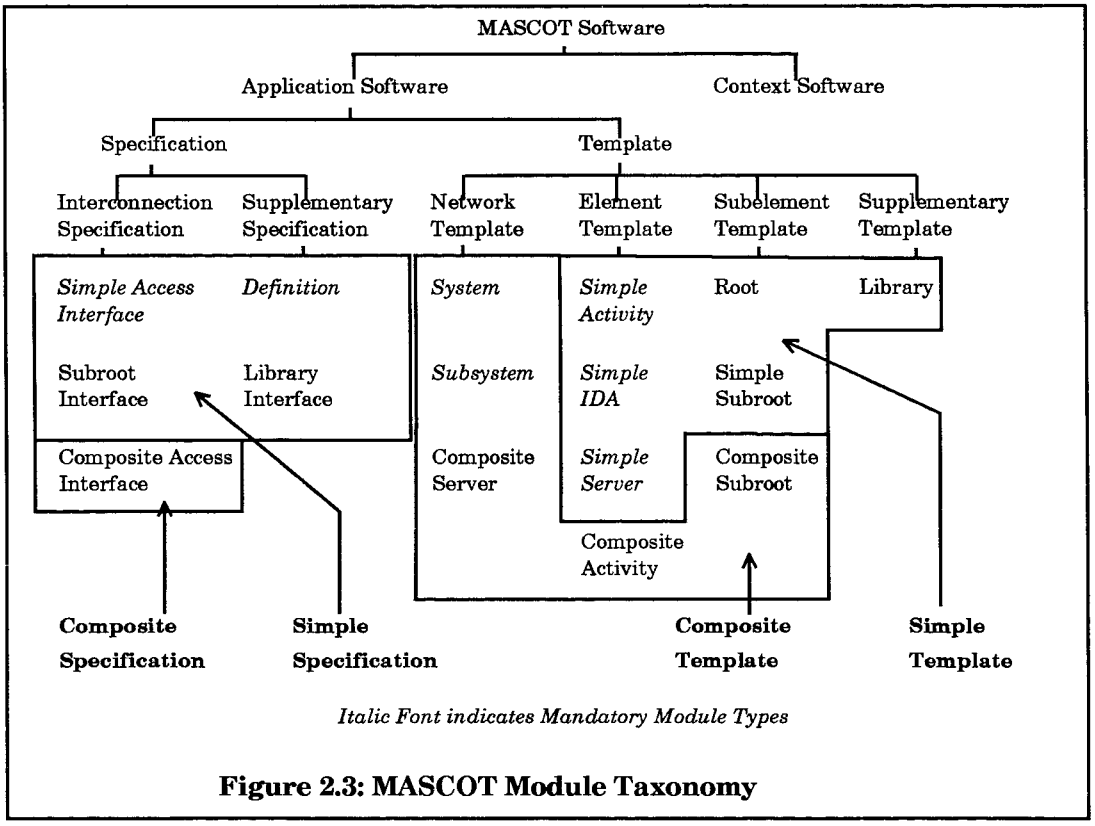


Figure 2.3: MASCOT Module Taxonomy

The primary structuring mechanisms used in MASCOT-3 are subsystems, composite components, and templates. Subsystems are arbitrary (well-formed) fragments of a MASCOT-3 network. Subsystems may be nested within other subsystems, thus forming a hierarchy.

Composite components include composite servers, IDAs, access interfaces, and activities. Composite IDAs may contain networks of composite or simple IDAs, and composite servers may contain networks of servers and IDAs. Composite access interfaces enable a number of paths between components to be combined, and presented graphically as a single line. This simplifies the pictures corresponding to MASCOT-3 designs and, used to reflect suitable abstractions, can keep a design comprehensible. Composite access

interfaces, like ordinary access interfaces, are known as interface specification modules, they define the interfaces between the components of a MASCOT-3 design.

Subsystems, and composite servers, IDAs, and access interfaces, do not have any semantic content. That is, they do not describe entities whose operation influences the behaviour of the design. They enable large design networks to be partitioned and presented hierarchically.

Composite activities are used in MASCOT-3 in the place of simple activities when the internal sequential data-flow of the activity is to be defined. This data-flow is defined in terms of instances of sub-element templates (see Figure 2.3). This data-flow description itself may be presented hierarchically, using composite subroots. MASCOT activities cannot, unlike for example CSP processes, be decomposed into a network of active components.

Elements of a MASCOT-3 design are partitioned into two broad kinds: specifications and templates.

Composite servers and subsystems are instances of network templates. Multiple instances of a template may occur in the same design, thus easing design re-use. Templates may not be defined recursively. Network templates are used in structuring and presenting a MASCOT-3 design, but they do not have a semantics independent of the components they contain.

A MASCOT design is hierarchical. At its top level is a *system template*. An implementation of the design is said to be an instance of this template.

For a more detailed introduction to MASCOT-3 the reader is referred to [JIM87].

2.5. The Graphical Notation

A MASCOT-3 design has both a graphical and a textual representation; however the graphical representation cannot portray all of a design. In particular, the type and procedural information associated with MASCOT specifications is not portrayed graphically using the MASCOT-3 notation. Types and procedures are considered to be defined outside of the MASCOT-3 design notation; usually in the programming language chosen to implement the design.

The MASCOT graphical notation follows a number of conventions. It is described here, and illustrated in Figure 2.4, below.

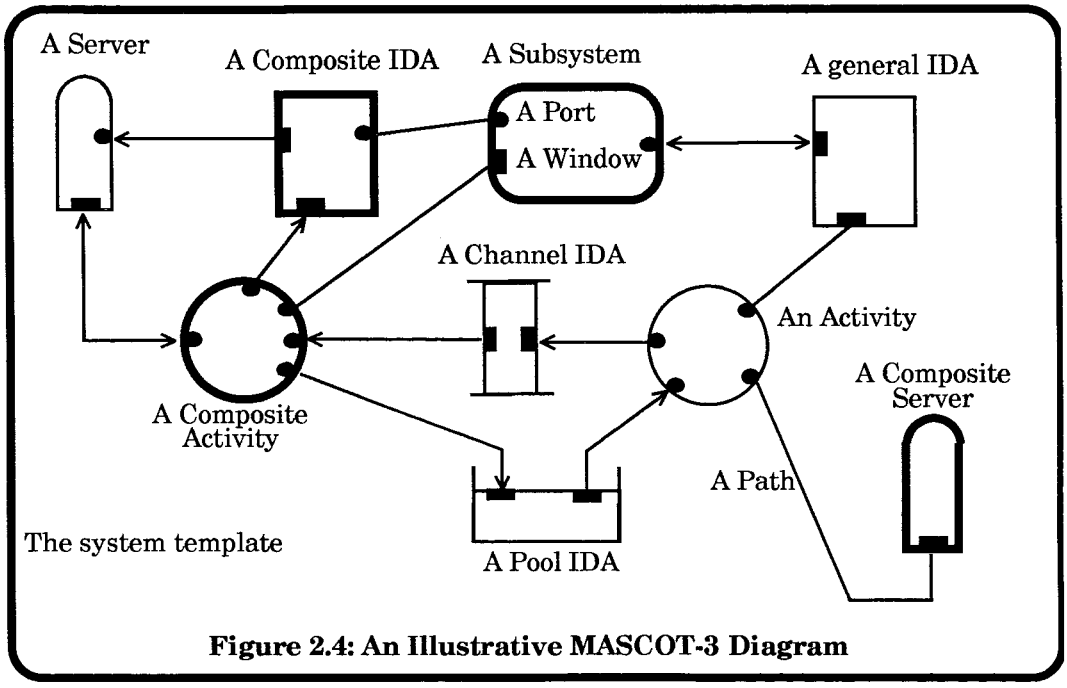


Figure 2.4: An Illustrative MASCOT-3 Diagram

Active, or potentially active, components are portrayed with rounded edges. For example, activities are represented as circles; servers are represented as boxes with a semi-circular end; and subsystems, which may contain activities or servers, as rectangles with rounded corners. Non-active components, such as IDAs, are represented by symbols drawn with straight lines. For example, a general IDA is a box; a pool IDA is a rectangle whose short edges protrude past one long edge; and a channel IDA is a rectangle whose short edges protrude past both long edges. Composite components are indicated by thicker edges; and their contents are presented on separate diagrams.

Ports are drawn on an inside edge of a component as a small filled circled. Windows are drawn on the inside edge of a component as a small filled rectangle.

Paths between components are represented as lines. The access interface associated with a window or port is mentioned as a label beside the line which represents the path that connects to that window. The direction of data-flow along a path may be represented by an arrow an the end of the line. Bi-directional data-flow may be represented by arrows at both ends.

Not all the MASCOT-3 graphical notation has been illustrated in the MASCOT-3 diagram above; omissions include: composite ports and windows; roots; subroots; and subroot interfaces. For full details see [JIM87].

2.6. The MASCOT-3 Method

MASCOT-3 claims to be much more than simply a design notation. It also claims to address the design process; software construction; software operation; and software testing. It furthermore claims to support design re-use. The MASCOT-3 method will be briefly described and evaluated using these categories.

2.6.1. Deriving the Design

This is not a subject which the MASCOT method addresses in any depth. The MASCOT handbook, [JIM87], explains that: "[MASCOT-3] does not provide recipes for filling out the framework for a given design problem". Not only is MASCOT not prescriptive, but the handbook contains no informal guidelines for developing MASCOT designs.

Developing MASCOT designs is an incremental process. MASCOT allows a progressive consideration of a system's functionality through abstraction due to hierarchical facilities of the design notation. MASCOT designs may be developed in a top-down or bottom-up manner, or a mixture of both.

The lack of rules or recipes for design derivation should not be seen as a failure of MASCOT as a design method. The lack simply reflects the fact that design is a creative process, and is currently inadequately understood. It is arguable that it is better to provide the designer with a flexible and appropriate language in which to express designs naturally, than to guide, perhaps inappropriately.

One element of the MASCOT notation that particularly recognises that designs are developed incrementally is a "status" assigned to design elements. There are five status values: Registered; Partially and Fully Introduced; and Partially and Fully Enrolled. The conditions under which an element may be ascribed a particular status are defined recursively over the structure of a MASCOT design. An element's status reflects how completely it has been defined, and the status of the other elements it depends upon. The precise meaning of each status value can be found in [JIM87].

2.6.2. Software Construction

MASCOT-3 does not define how to develop software to implement a MASCOT design. However, software constructed from a MASCOT-3 design should reflect the structure of the design.

Software constructed from a MASCOT-3 design is more likely to reflect the component, rather than the template, structure of the design, unless the implementation language supports a form of generics. Network templates and their instances may have no mapping into the implementation.

Clearly MASCOT only provides the outline of a software implementation. Even a composite activity whose internal data-flow has been defined may be implemented to perform many different computations. It is this ambiguity which will be removed from MASCOT when designs are given a formal semantics in Chapter 5.

2.6.3. Software Operation

MASCOT-3 is less prescriptive than the previous version of MASCOT, (i.e. MASCOT-2 [JIM83]), concerning what facilities it requires the execution environment to provide. This allows MASCOT designs to be implemented in a number of different languages and run-time systems. MASCOT-2 defined a set of scheduling primitives (commands) that the kernel (run-time system) had to provide for use in the activities and IDAs. MASCOT-3 defines no such list of obligatory primitives.

An implementation only needs to reflect the structure of the design during execution. This means that activities are implemented as parallel processes, which can only communicate via IDAs, along the paths given in the design. However, MASCOT-3 does not define whether an implementation is to be concurrent on a single processor; distributed on separate processors; or a mixture of both. Nor, for distributed processors, does it prescribe local or global kernels. This ambiguity adds to the generality of MASCOT, but reduces what may be deduced or proven about a MASCOT design.

2.6.4. Software Testing

MASCOT-3 claims to provide facilities which ease the task of software testing. However, the introduction to [JIM87] indicates that the "testing facilities" are actually a

by-product of representing the design as de-coupled components. The MASCOT design notation forces the interfaces between the various components of the design to be explicitly identified. This eases the task of developing test harnesses for parts of an implementation of a MASCOT design. The ports and windows of a MASCOT component (for example, a subsystem) define the access interfaces that the component expects to see and provide. The system around such a component can be replaced by test harness components, providing they provide and expect the same interface.

MASCOT-3 does not provide recipes describing how best to test a MASCOT design.

2.6.5. Design Re-Use

Design re-use in MASCOT is a result of three properties of the MASCOT-3 design notation: templates; access interfaces; and subsystems. Templates may be multiply instantiated within one design, or may be instantiated within more than one design, thus supporting re-use. Access interfaces, by providing a well-defined interface, ease the re-use of components in different designs. Subsystems encourage the partitioning of designs, hopefully into re-usable components. MASCOT-3 however provides no guidelines on what constitutes a good or easily re-usable component.

2.6.6. An Evaluation

Undoubtedly, MASCOT's strength lies in its approach to design representation. The MASCOT-3 design notation allows a design's concurrent agents, and the data flows between them, to be explicitly represented. Its hierarchical presentation, and its strong emphasis on the de-coupling of components by IDAs and the strong typing of the interfaces between components, eases the design task. The IDA communication model is richer than that found in many concurrent languages, such as Occam, [INM88], Ada, [LRM83] and CSP, [Hoa85], which only support synchronous communication. It supports message passing, like CSP, and shared variable communication, like extended VDM, [Jon83]. This richness of the IDA communication model adds flexibility to the design. The appropriate protocol may be adopted for each part of the design.

The asynchronous pool IDA, in particular, enables designs to be constructed which are robust against component failure. MASCOT has proven itself a suitable design notation for defining the structure of large, robust, real-time, embedded software products.

The extent to which MASCOT-3 method addresses the design, construction, operation, test, and re-use elements of software development is variable, and is usually a by-product of the design notation. However, its philosophy on these subjects seems adequate.

MASCOT has been favourably reviewed in the following comparisons of design methods for real-time systems: [HOH91a] and [Jac90]. However, there are four significant areas where MASCOT may be criticised. Firstly, the design notation does not support the recording of temporal information. However, currently it is not clear what temporal information ought to be recorded. There is some ongoing research in this direction, for example the SPIRITS[18] project, [IED91]. Secondly, MASCOT only captures the architecture of the design and not its full functionality. This is normal for design methods, and many designers find this abstraction helpful when considering a design at an early stage. However, it does prevent the correctness or safety properties of the design being established before the software is developed. Thirdly, the MASCOT design notation does not have a widely used formally defined semantics[19]. Fourthly, MASCOT has failed to create a large user-base outside of the UK defence industry.

MASCOT has also been criticised for requiring a large number of names to be generated to label a design. However these are needed to present a design unambiguously, [HOH91a].

Personally, I consider the core MASCOT approach to real-time software design of hierarchically presenting such designs as static networks of activities strongly de-coupled by IDAs to be one of the best currently developed. DORIS, [Sim93], and MOON, [HOH91b], have both demonstrated how MASCOT can be usefully extended, while retaining this same design philosophy.

[18] Supporting Predictable Implementation of Requirements on Time and Safety

[19] [BJP87] is a description of a previous attempt to define the semantics of MASCOT-3 designs. A fuller discussion of [BJP87] can be found in chapter 7.

Chapter 3: Formalising MASCOT

This chapter addresses the formalisation of the MASCOT design method. It starts by considering how the various elements of a design method can be formalised, and the objectives of such formalisation. This is followed in section 3.2 by an introduction to node labelled controlled graph grammars. These are used in subsequent sections to define the abstract syntax of simplified MASCOT designs. The MASCOT designs considered are a significant simplification of MASCOT-3 designs, but they retain most of the operational expressiveness of MASCOT-3. The abstract syntax defined in this chapter is used in subsequent chapters to structure the definition of a denotational semantics of MASCOT designs.

3.1. Formalising Design Methods

The formalisation of design methods is considered in this section. Section 3.1.1 contains a discussion of the parts of a design method that can be formalised, and section 3.1.2 analyses the benefits of formalising design notations.

3.1.1. Formalisation

When discussing the formalisation of a design method it is important to be clear about what part or element of the design method is being formalised. The elements found in many design methods include: the rules for using the method to produce designs; the syntax rules of the design notation; and the semantics of a design written in that notation. A more intricate model of a design method is given in [HoW92], and is repeated below in Figure 3.1. It distinguishes five elements of a design method: the *way of thinking*; the *way of control*; the *way of modelling*; the *way of working*; and the *way of support*.

According to [HoW92], the *way of thinking* is the basic assumptions and view point of the method; the *way of control* concerns the project management elements of the method; the *way of modelling* is the models and model components used in the design; while the *way of working* is the strategies and procedures of the method for arriving at the models. The *way of support* deals with tool support prescribed by the method.

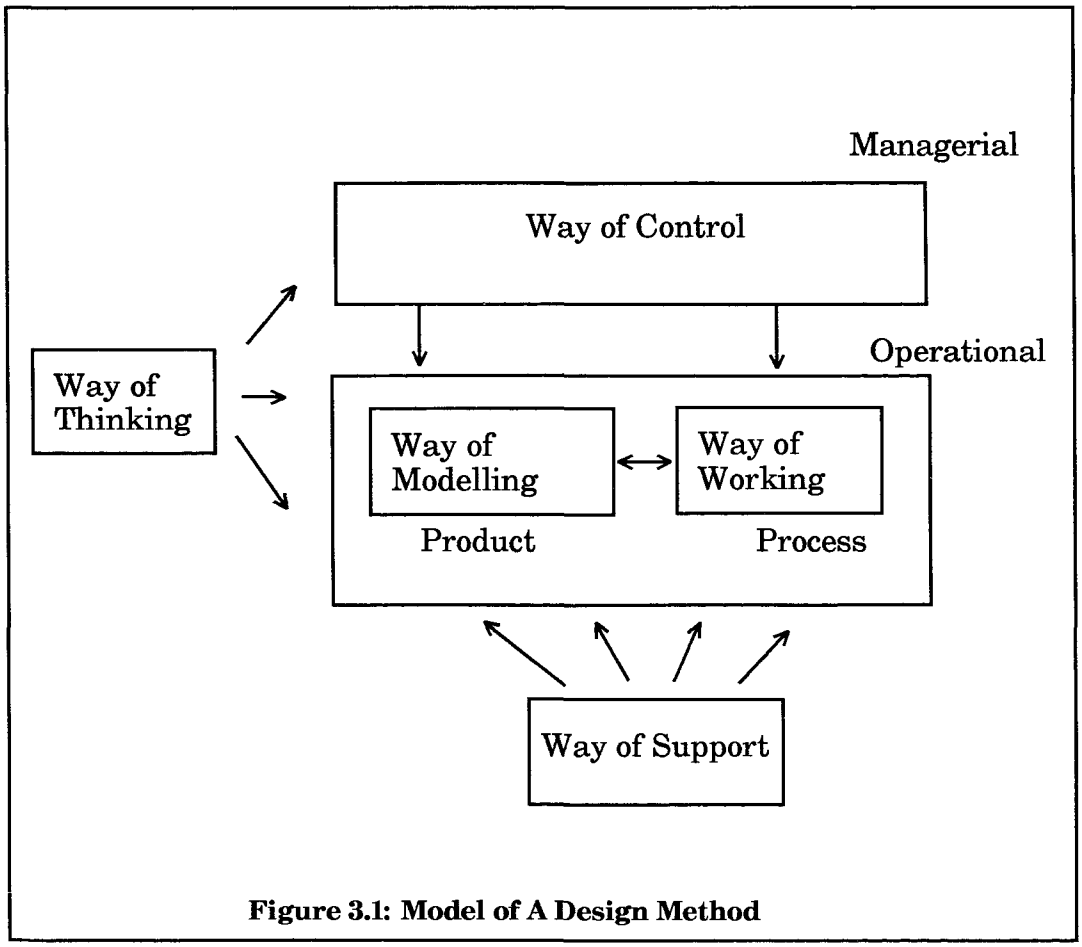


Figure 3.1: Model of A Design Method

Rather than considering this model of a design method in a vacuum, each element will be related to the MASCOT design method. The formalisation of each element is briefly discussed.

3.1.1.a. The Way of Thinking

The *way of thinking*, or the basic assumptions and view point of a design method, are unlikely to be clearly identified, as they are often shared and unquestioned by the originators of the method. However, clearly, such elements are fundamental, and are likely to have a significant impact on the applicability of the method for different types of software. This information is usually passed on to the users of the method through general statements and the presentation of a number of small examples.

MASCOT is directed towards the design of large embedded systems, with distributed or concurrent software. It has a hierarchical and network or data-flow

view of software systems. However, beyond this, clearly delineating its *way of thinking* is beyond the scope of this thesis.

It is not clear how the underlying world view and philosophy of a design method could be formalised. However, any method will benefit from having as clear a presentation of this element of it as possible.

3.1.1.b. The Way of Control

The rules and guidelines concerned with the project management of designs comprise a method's *way of control.* There are three main elements of MASCOT which could be considered its way of control: the identification of "technical authorities"; a MASCOT database which records and controls the status progression of the elements of the design; and it documentation requirements.

MASCOT-3 requires "technical authorities" to be responsible for defining the requirements for MASCOT templates and (informally) verifying and accepting the designer's design of the template. MASCOT-3 requires a Requirements Review; and a Verification and Acceptance Review, to monitor the transfer of work between technical authority and designer. This is clearly part of MASCOT's *way of control.*

In MASCOT-3 each component of a design is ascribed a status which may be one of the following: registered; partially introduced; (fully) introduced; partially enrolled; and (fully) enrolled. A component can only be progressed to a higher status if it satisfies certain preconditions; preconditions related to its completeness and consistency. For example, the preconditions for a composite component require all the elements of which it comprises to have acquired a certain status before the composite component can be progressed to a higher status. Details can be found in [JIM87]. It may be debated as to whether this is really part of MASCOT's way of control, or whether such status progression rules are part of the static (syntactic) constraints on MASCOT designs, and hence part of its *way of modelling.*

The MASCOT handbook, [JIM87], also provides a helpful discussion of design documentation and how it ought to be maintained, preferably using a configuration management database. However, MASCOT is not prescriptive concerning design documentation, and so it could be argued, that this element of MASCOT's *way of control* need not be formalised.

It ought to be possible to formalise a design method's *way of control*. A model of the process of design when using the method can be formally defined, as can the inputs required (e.g. the authorisations and previously finished tasks) to move from one phase to the next, and the outputs (e.g. the reports, documents, or justifications) for each phase. Formalisation of the *way of control* of a method would undoubtedly clarify exactly what a properly controlled design process meant. It would force the relationship between different reports and authorisations to be made unambiguous, so that a given development could be checked for compliance with the method. The objective of such formalisation would be to clarify the process, and perhaps to aid the development of a configuration tool.

No attempt will be made in this thesis to formalise MASCOT's *way of control*.

3.1.1.c. The Way of Support

MASCOT does not prescribe particular tool support, so, strictly, MASCOT has no *way of support* element. However, this is not to say that tools have not been produced which support MASCOT, for example the BAe MADGE tool, [Har91].

There would not seem to be much scope in formalisation of the *way of support* of a design method. A clear statement of which tools should be used should suffice, if a method is prescriptive on this issue. If a method hints at the properties that a tool that supports the method should exhibit, undoubtedly these would benefit from being specified formally. Presumably this would exploit existing formal specification languages. The objective of such formalisation would be to state precisely the behaviour of the tools so that tools developed by different vendors would provide a guaranteed functionality.

3.1.1.d. The Way of Working

The *way of working* of a design method is the process for developing designs using that method. However, MASCOT is not a prescriptive design method, in that it does not tell one how to arrive at a MASCOT design. The MASCOT-3 Handbook, [JIM87], page 5-2, claims "MASCOT is not prescriptive: thus it does not provide any recipes for establishing or filling out the framework for a given design problem"

It can be argued that the *way of working* concept ought to be extended for MASCOT to include the construction of software (the "shell" of a program) from the design. MASCOT requires that software has the same structure as its design. However, this is not part of MASCOT that will be considered further in this thesis.

Formalisation of the *way of working* will involve developing the process model mentioned above in the discussion about the formalisation of the *way of control*. Formalisation of the *way of working* may also consist of a precise statement of how a design is to be derived.

3.1.1.e. The Way of Modelling

The *way of modelling* is the models and model components used in the design. The *way of modelling* is perhaps the most important element of a design method to formalise. It entails defining what a valid design is, and what it means. It should include a definition of the concrete and abstract syntax[20] of the design notation, and a definition of the semantics of each element of the design, including the semantics of any interconnections or interfaces which limit interactions between the elements. Many design methods use more than one notation or diagram to record design information, a full formalisation of such a design method would necessitate the precise relationship between these different representations of the design to be clarified, and formally stated.

In formalising a design method's *way of modelling* it is likely to be a trade-off between capturing the ambiguity in the informal description of the design and changing the semantics of components so that they have a clean model. In this thesis the trade-off is resolved in favour of changing or clarifying the semantics. It is believed that this is the best hope of developing a semantics which can be utilised in formally developing industrial scale software.

MASCOT's *way of modelling* is its design notation, both graphical and textual; along with the semantics of the various elements of a design: activities; IDAs; paths servers; access interfaces; subsystems; and templates etc.. The concrete syntax of

[20]The word "syntax" is used here, as in [HoW92], to include the definition of the notation of the design representation and the rules which restrict which designs are considered well-formed, or valid. "Semantics" is used to refer to the meaning of well-formed designs.

the MASCOT textual notation has been formalised using a modified BNF[21] in [JIM87]. This fails to capture the well-formedness rules of MASCOT, which are only presented informally in [JIM87]. More general languages, such as GDL, [SWB87], and [WBS90], have been proposed for describing the graphical syntax of design languages, including positioning and well-formedness rules. However, instead of using GDL, graph grammars will be used in later sections to define the abstract syntax for a subset of MASCOT-3. Developing a formal semantics for MASCOT is the topic of chapters four and five.

3.1.1.f. Summary

The formalisation of design methods has been discussed with specific reference to MASCOT-3 using the model given in [HoW92]. It was concluded that a method's process model could be formalised; as could the specification of any prescribed tools and the rules for constructing designs. However, the most important element of a design method to formalise is its *way of modelling*.

3.1.2. The Benefits of Formalisation

In this section the benefits of formalising the *way of modelling* element of a design method are considered. Benefits which have been suggested in the literature are presented. The benefits are broken down into two lists: those associated with formalising the syntax of the design representation; and those associated with formalising the semantics of designs.

The semantics of design notations are usually insufficient to define the full functionality of a design. Therefore, in the process of being formalised, the syntax and semantics of the design notations may be extended to rectify this, for example [MVL92]. Therefore, some of the literature talks of integrating design and formal methods, rather than formalising design methods, for example [TKP90] and [SFD92]. The pre-existing semantics of the design method are usually formalised during such integration.

[21] Bachus-Naur Form

3.1.2.a. Design Notation Syntax

It has been contended that the following benefits arise from formalising the syntax of a design notation.

- It facilitates the automatic checking of the syntactic consistency of the design. This includes:
  - the detection of syntactic discrepancies between different parts of the design; similar to the compilation process for programs.
  - the checking of the well-formedness of the design hierarchy, and the detection of incomplete designs.
- Easier construction of software tool support, [Tse91].
- Data flows in the design can be checked for "continuity": [Tse91]. This consists of analysing the data and information flow of a design (assuming the design method enables that to be recorded) and checking that inputs affect outputs, and data is not generated from nothing.

3.1.2.b. Design Notation Semantics

It has been argued, [SrH85], [TKP90], [Tse91], [HoW92], and [SFD92] that the following benefits arise from formalising a design notation's semantics:

- The removal of ambiguity in the meaning of a design, facilitating communication between designer and specifier, and designer and programmer.
- The ability to establish the equivalence of different representations of the design.
- The ability to prove the implementation correct with respect to the design, assuming the implementation has a formal semantics.
- Properties of the design may be able to be proved.
- The proliferation of formal techniques and correctness issues in industrial software development.

- The ability to integrate different design methods (if they both have a formal semantics), and to compare or transport designs between them.
- The facilitation of the creation of sound prototyping tools.
- Formal specification languages gain a graphical presentation notation, and maybe modularisation constructs.

The realisation of at least some of these benefits is not automatic upon formalisation of a design notation. To some extent it depends upon the quality of the semantic models defined. If the semantics are to be used they must be accessible to the users of the method, and must be neither too complicated, nor too great a revision of the informal intuitions of the method. Furthermore, some of the benefits only arise if the design notations are extended so that they provide a more complete description of the functional behaviour than is common for design notations.

3.2. An Introduction to Graph Grammars

This section introduces and provides the basic definitions of graph grammars needed in this thesis. Graph grammars are used in subsequent sections to define the abstract syntax of the MASCOT designs which are to be given a formal semantics. The set of valid MASCOT designs was characterised graph theoretically in section 2.3. However, it is difficult to structure and formalise the definition of the denotational semantics of designs defined graph theoretically. Semantic definitions of programming languages, ([Pag81], [Sch86], [Hen90], and [NiN92]), are usually structured around the abstract syntax of the language. However, most programming languages are textual and one-dimensional, hence string grammars are adequate for defining their abstract syntax. MASCOT designs, however, are essentially two-dimensional, and their abstract syntax is better defined using graph grammars[22]. This is the motivation for introducing graph grammars in this section.

[22]MASCOT-3 has a textual representation which obviously could be given an abstract syntax using a string grammar. However, it is contended that the *structure* of the textual form of a MASCOT design is not as revealing as the *structure* of the graphical form of a MASCOT design. The structure of the textual form of a MASCOT design is a collection of elements, the details of which reveal their inter-relationship, whereas the structure of the graphical form of a MASCOT design includes this inter-relationship information. It is further contended that the inter-relationships between the elements of a design is fundamental for understanding the design, and that a semantics based around this information will be more transparent. The complexity of graph grammars over string grammars is hence worth accepting in this case.

Graph grammars are a generalisation of string grammars, where strings are replaced by graphs. However, graph grammars are significantly more complicated than string grammars, there are numerous variations, and the supporting body of theory is still under development, [Kap90]. A good overview of the field can be obtained from the proceedings of the four international workshops on graph grammars: [CER79]; [ENR83]; [ENR86]; and [EKR90].

Like string grammars, graph grammars can be characterised as context free or context sensitive. However, unlike string grammars, the non-terminal symbols of graph grammars may represent more than simply sequences of terminal and non-terminal symbols. That is, a graph grammar's productions may define rules which replace a non-terminal symbol with, for example: subgraphs; edges; hyperedges; or nodes. Graph grammars also differ from string grammars in that there are a number of options concerning how the replaced graph is embedded in the remaining graph. The grammars which are introduced and used in this chapter are from the class of context-free graph grammars known as Node Labelled Controlled (NLC) grammars. These only allow node replacement with local embedding conditions. This will be explained below.

3.2.1. Terminology

Basic graph theory terminology has been introduced in section 2.3.1. This section introduces further terminology to ease the discussion of graph grammars.

A *node labelled graph* is a four-tuple (N, A, Σ, L), where N is the finite set of nodes; and $A \subseteq N \times N$, is the set of arcs; Σ is the set of node labels or *alphabet* of the graph, and $L: N \rightarrow \Sigma$ is the *labelling function*. L(n), where $n \in N$, is the label of node n. (This definition is standard, but was taken from [Roz86].) An *edge labelled graph* is a four-tuple (N, A, Σ, L), where N is the finite set of nodes; and $A \subseteq N \times N$, is the set of arcs; Σ is the set of edge labels or alphabet of the graph, and $L: A \rightarrow \Sigma$ is the labelling function. L(a), where $a \in A$, is the label of arc a. A *node/edge labelled graph* is a six tuple ($N, A, \Sigma_N, \Sigma_E, L_N, L_E$), where N is the finite set of nodes; and $A \subseteq N \times N$, is the set of arcs; Σ_N is the set of node labels; Σ_E is the set of edge labels; $L_N: N \rightarrow \Sigma_N$ is the node labelling function; and $L_E: A \rightarrow \Sigma_N$ is the edge labelling function.

A (sequential) graph grammar consists of a set of production rules, and an initial graph. A production rule defines the removal of a subgraph, known as the *mother graph*, from the *host graph*, the replacement of the mother graph with a *daughter graph*, and

the embedding of the daughter graph in the host graph with the mother graph removed. The host graph with the mother graph removed is sometimes called the *restgraph*.

Where one graph can be obtained from another by one application of a production rule from a graph grammar, that graph is said to be a *direct derivation* of the second in that grammar. A graph that can be obtained from another by a sequence of production rule applications is said to be a *derivation* of the second graph.

Where the mother graphs are always single nodes, and the embedding definition is always in terms of nodes in the host graph which are in the neighbourhood of the mother node, the grammar is called Node Labelled Controlled (NLC).

3.2.2. NLC Graph Grammars

The following definition is standard, but was taken from [EnR90]. A NLC graph grammar, GG, is a five-tuple: (Σ, Δ, Z, P, C), where Σ is the set of all node labels, Δ is the set of terminal node labels, Z is the initial node labelled graph over Σ, P is the finite set of production rules, and C is the embedding relation. $\Sigma - \Delta$ is the set of non-terminal node labels. A production rule has the form: $x ::= Y$, where $x \in \Sigma - \Delta$, and Y is a graph with node labels from Σ. The embedding relation, C, is a binary relation over Σ, that is, $C \subseteq \Sigma \times \Sigma$. Each member of C, (a, b), is known as connection instruction, and defines that, once a production rule $x ::= Y$ has been applied to a node in the host graph labelled with non-terminal x, every node labelled with a in the daughter graph Y is connected to every node in the restgraph in the neighbourhood of the mother node labelled with b.

For example, with initial graph Z, embedding C, and rule R1 shown in Figure 3.2, the direct derivation is the final graph shown.

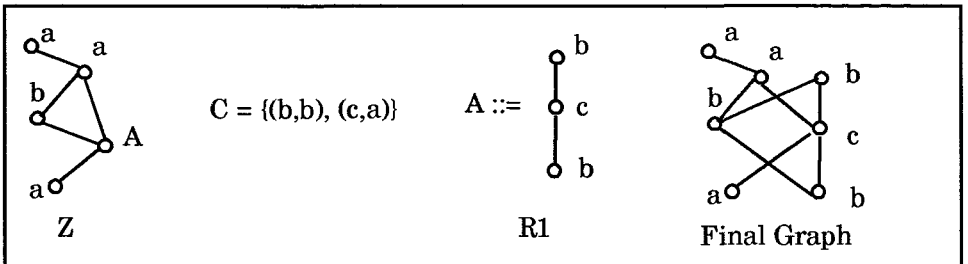


Figure 3.2: An NLC Grammar and Derivation

The graph language generated by GG is $L(GG) = \{ g \in GR(\Delta) \mid Z \Rightarrow^* g \}$, where $GR(\Delta)$ is the set of graphs labelled with symbols from Δ, $\Rightarrow$ represents a direct derivation, and $\Rightarrow^*$ represents a derivation.

The embedding relation defined above is not expressive enough to define the abstract syntax of MASCOT designs, and so NCE graph grammars are introduced in the next section.

3.2.3. NCE Graph Grammars

Basic NLC graph grammars only allow daughter graphs to be embedded in hosts by reference to the labels of the nodes of the daughter graph. Neighbourhood Controlled Embedding graph grammars, [EnR90], allow the actual daughter nodes to be referenced, not just their labels.

A NCE graph grammar is a four-tuple: (Σ, Δ, Z, R), where Σ, Δ, and Z are as for NLC graph grammars, and R is the finite set of re-writing rules, each rule taking the form: $(x ::= Y, C)$, where $x ::= Y$ is the production, and C is the connection relation for Y, $C \subseteq V_Y \times \Sigma$. V_Y is the set of nodes of Y.

The embedding relation in NCE grammars is local to each production rule, and so is much more expressive than the global one used in basic NLC grammars.

For example, with initial graph Z, and rule R1 shown in Figure 3.3, the direct derivation is the final graph shown.

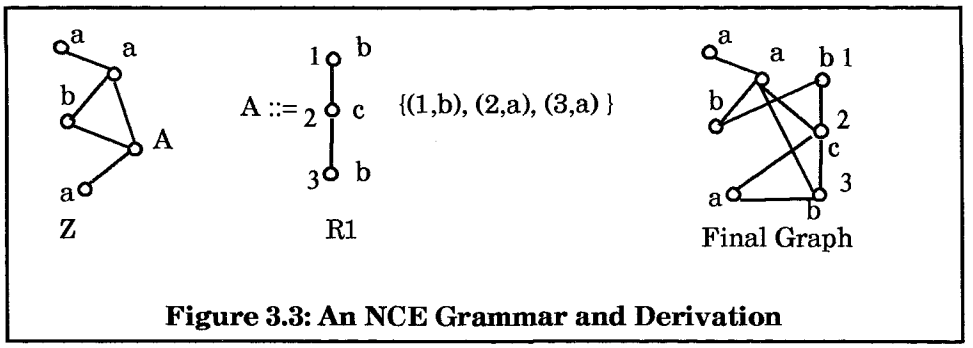


Figure 3.3: An NCE Grammar and Derivation

3.2.4. dNCE Graph Grammars

The basic NCE graph grammar definition above is extended in this section, as in [EnR90], to provide distinguishable embedding connections between host nodes

depending upon the direction of the arc between the host node and the mother node. Again, this complexity is needed in defining a graph grammar for the abstract syntax of MASCOT designs.

A directed NCE (dNCE) graph grammar is a four-tuple: (Σ, Δ, Z, R), where Σ, Δ and Z are as for previous graph grammars, except that Z is a directed graph. R is the finite set of re-writing rules, each rule taking the form: ($x ::= Y, C_{in}, C_{out}$), where $x ::= Y$ is the production; and C_{in} is the connection relation for the incoming connections to the nodes of Y, $C_{in} \subseteq V_Y \times \Sigma$; and C_{out} is the connection relation for the outgoing connections from the nodes of Y, $C_{out} \subseteq V_Y \times \Sigma$. As before, V_Y is the set of nodes of Y.

3.2.5. deNCE Graph Grammars

The final extension which is considered enables the embedding connection to distinguish between nodes in the host graph. Generally, these nodes cannot be referred to directly due to the arbitrary number of the nodes in the neighbourhood of the mother node, instead the nodes are partitioned according to the label on the arc connecting the host node to the mother node. Such grammars are called edge labelled NCE grammars (eNCE grammars). An embedding connection instruction is a four-tuple (x, q, m, p), where x is a daughter node, p and q are edge labels and m is a node label. It should be interpreted as connecting, using an arc labelled with a "q", a daughter node "x" to all nodes labelled with an "m" which were connected to the mother node with an arc labelled with a "p".

Strictly, eNCE grammars do not need the directed extension used in dNCE grammars as the direction information can be coded in the edge labels. However, it is considered clearer to maintain this separation, resulting in grammars known as deNCE graph grammars, [EnR90].

A deNCE graph grammar is a four-tuple: (Σ, Δ, Z, R), where Σ and Δ are as for previous graph grammars; Z is the initial node/edge labelled graph; and R is the finite set of re-writing rules, each rule taking the form: ($x ::= Y, C_{in}, C_{out}$), where $x ::= Y$ is the production; and C_{in} is the connection relation for the incoming connections to the nodes of Y, $C_{in} \subseteq V_Y \times L \times \Sigma. \times L$; and C_{out} is the connection relation for the outgoing connections from the nodes of Y, $C_{out} \subseteq V_Y \times L \times \Sigma \times L$. As before, V_Y is the set of nodes of Y. L is the set of edge labels.

For example, with initial graph Z, and rule R1 shown in Figure 3.4, the direct derivation is the final graph shown.

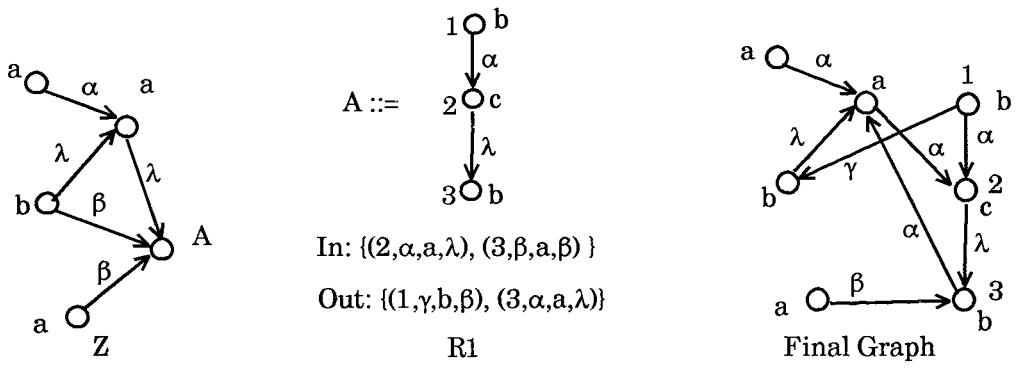


Figure 3.4: A deNCE Grammar and Derivation

3.3. A deNCE Graph Grammar for Simple Linear MASCOT Designs

In this section a deNCE graph grammar is presented which defines the abstract syntax of particularly simple MASCOT designs. The set of MASCOT designs defined is given by the language of the graph grammar.

A deNCE graph grammar for Simple Linear MASCOT (SLM) is SLM_GG = (Σ, Δ, Z, R), where Σ, the set of all node labels, is {A_IDA, IDA, pool, channel, signal, activity}, Δ, the set of all terminal symbols, is {pool, channel, signal, activity}, and Z, the initial node/edge labelled graph, is[23] ({i', ai'}, {(i', ai')}, {A_IDA, IDA}, {read'}, {(i', IDA), (ai', A_IDA)}, {((i', ai'), read')}). Z can be presented pictorially as in Figure 3.5, where nodes are drawn as boxes,

[23]Remember the signature of node/edge labelled graphs is: $(N, A, \Sigma_N, \Sigma_E, L_N, L_E)$.

the name of the nodes adjacent to the box, the label of the node in the box, arcs as arrows between the boxes, and arc labels adjacent to the arcs.

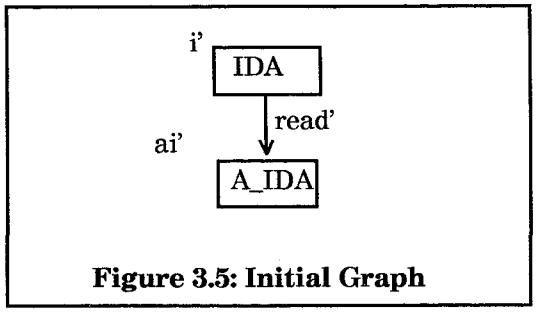


Figure 3.5: Initial Graph

R, the set of re-writing rules, is { r_1, r_2, r_3, r_4, r_5 }, where each rule, r_i, takes the form: (x ::= Y, C_{in}, C_{out}), where $x \in \Sigma - \Delta$, Y is an node/edge labelled graph, and $C \subseteq V_Y \times L \times \Sigma \times L$. The rules are defined below with the daughter graphs presented pictorially.

3.3.1. The Production Rules

r_1 = (IDA ::= ({p'}, $\emptyset$, {pool}, $\emptyset$, { (p', pool) }, $\emptyset$),
{ (p', write', activity, write*) },
{ (p', read', activity, read*), (p', read', A_IDA, read*) }).

The node name p' on the right hand side of r_1 is chosen on each application of the rule to be unique in the host graph. Likewise the edge labels read' and write' are also chosen to be unique, but prefixed by read or write respectively. The wild-card character "*" is used in the embedding definition to indicate that the given node links to any node (with appropriate label) connected by a link labelled with a value which starts with a read or write, irrespective of its unique suffix.

Pictorially, r_1 is:

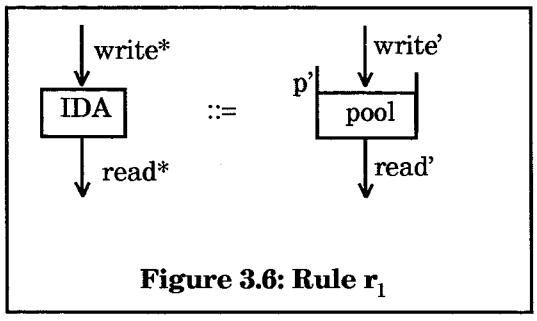


Figure 3.6: Rule r_1

It should be noted that this pictorial representation does not carry as much embedding information as the textual version of the rule.

r_2 = (IDA ::= ({s'}, ∅, {signal}, ∅, { (s', signal) }, ∅),
{ (s', write', activity, write*) },
{ (s', read', activity, read*), (s', read', A_IDA, read*) }).

Pictorially, r_2 is:

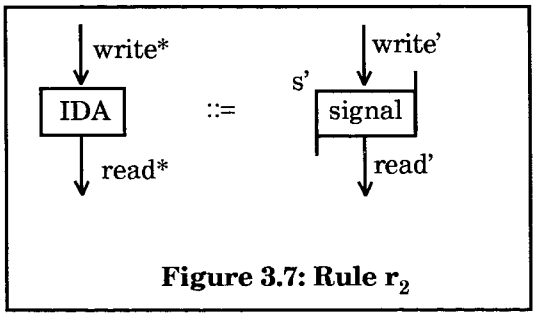


Figure 3.7: Rule r_2

r_3 = (IDA ::= ({c'}, ∅, {channel}, ∅, { (c', channel) }, ∅),
{ (c', write', activity, write*) },
{ (c', read', activity, read*), (c', read', A_IDA, read*) }).

The node name c' on the right hand side of r_3 is chosen on each application of the rule to be unique in the host graph.

Pictorially, r_3 is:

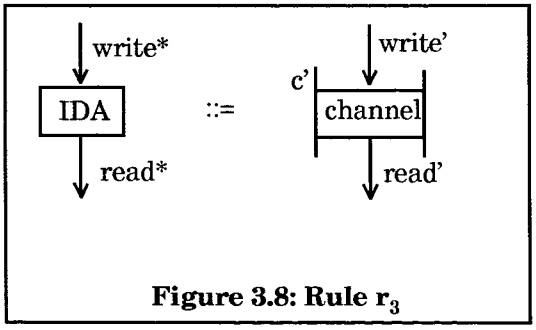


Figure 3.8: Rule r_3

r_4 = (A_IDA ::= ({ai', ai"}, { (ai', ai") }, {A_IDA}, {read'},
{ (ai', A_IDA), (ai", A_IDA) }, { ((ai', ai"), read') }),
{ (ai', read", channel, read*), (ai', read", pool, read*),
(ai', read", signal, read*), (ai', read", IDA, read*),
(ai', read", A_IDA, read*) },
{ (ai", read"', activity, read*), (ai", read"', A_IDA, read*) }).

Pictorially, r_4 is:

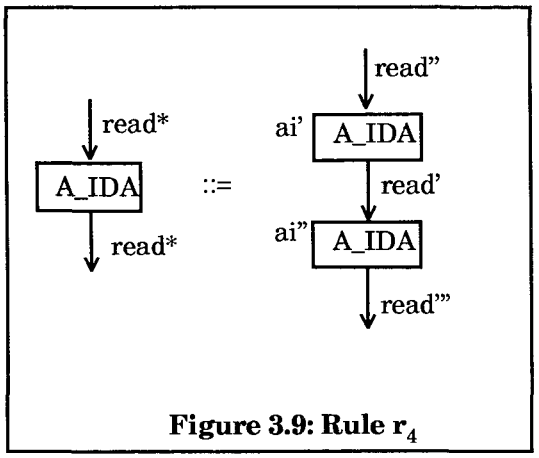


Figure 3.9: Rule r_4

r_5 = (A_IDA ::= ({a', i'}, { (a', i') }, {activity, IDA}, {write'}, { (a', activity), (i', IDA) },
{ ((a', i'), write') }),
{ (a', read', channel, read*), (a', read', pool, read*), (a', read', signal, read*),
(a', read', IDA, read*), (a', read', A_IDA, read*) },
{ (i', read", activity, read*), (i', read", A_IDA, read*) }).

Pictorially, r_5 is:

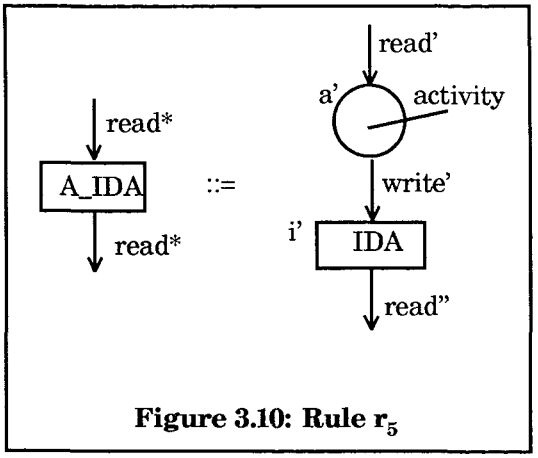


Figure 3.10: Rule r_5

The language of this grammar, L(SLM_GG), is the set of all simple linear MASCOT designs. This class of designs is one of the simplest.

3.3.2. An Example Design

Figure 3.11 contains is a valid derivation of a MASCOT design using the SLM graph grammar, where $\Rightarrow^x$ indicates the re-writing of the non-terminal node on the left hand side with the daughter graph associated with the production rule x.

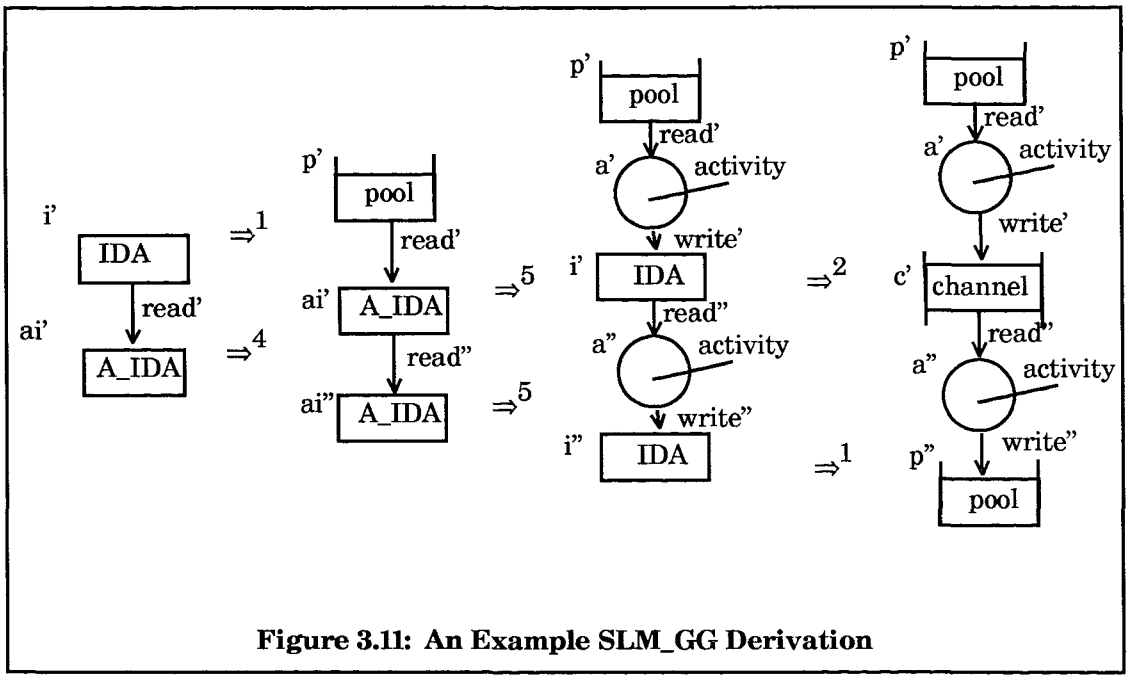


Figure 3.11: An Example SLM_GG Derivation

3.3.3. Ambiguity in the Grammar

Even with such a simple grammar as SLM_GG there can still be more than one sequence of production rule applications which will generate the same design. For example, r1, r4, r5, r5 r2, r1 and r4, r5, r2, r5, r1, r1 both generate the same design, the one shown in Figure 3.11. It can hence be argued that SLM_GG is an ambiguous grammar for simple linear MASCOT designs. However, it is contended that this is not a problem as SLM_GG is intended as an abstract grammar rather than as a concrete grammar.

It is normal in defining the syntax of programming languages, [Sch86], to separate the definition of the syntax into concrete and abstract syntaxes. The concrete needs to be unambiguous as it is responsible for defining the "derivation tree" for a given program. However, the abstract syntax may be ambiguous, as it defines the semantics of a program, *given* a particular derivation tree. Indeed, it is common for the grammar which defines a language's abstract syntax to be ambiguous, as ambiguous grammars are usually simpler, and they are usually clearer at revealing the structure of the language. The semantics of a language are defined over the abstract syntax of a language. SLM_GG is intended as an abstract syntax of MASCOT and so this ambiguity is unimportant.

3.4. A deNCE Graph Grammar for Branching/Looping MASCOT Designs

A deNCE graph grammar is presented informally in this section for Branching/Looping MASCOT designs, and is called BLM_GG. The Simple Linear MASCOT designs are a subset of such designs, and indeed the production rules of the SLM_GG graph grammar are also part of the BLM_GG grammar. These rules retain their original numbering. The grammar is presented formally in Appendix Six.

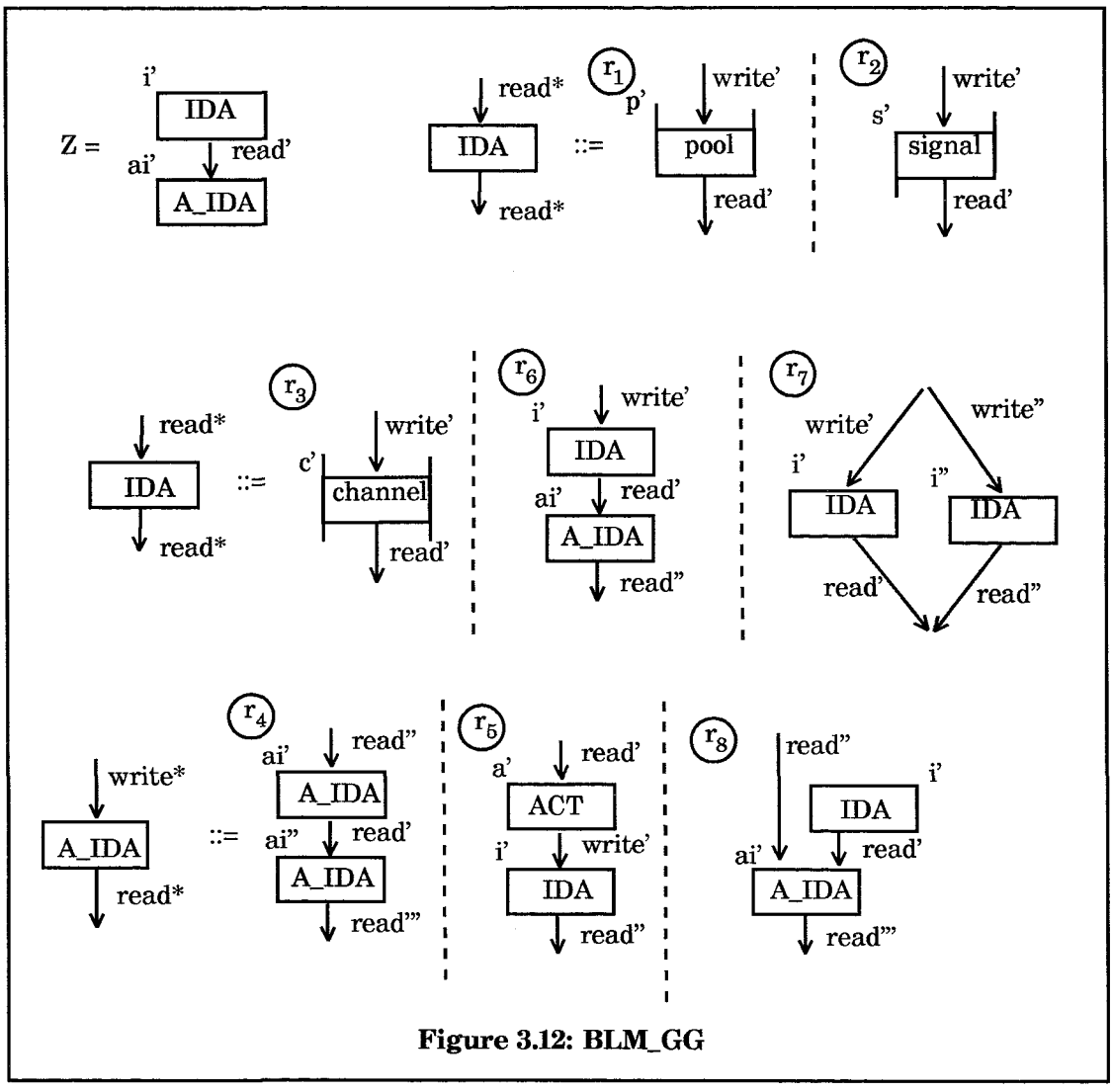


Figure 3.12: BLM_GG

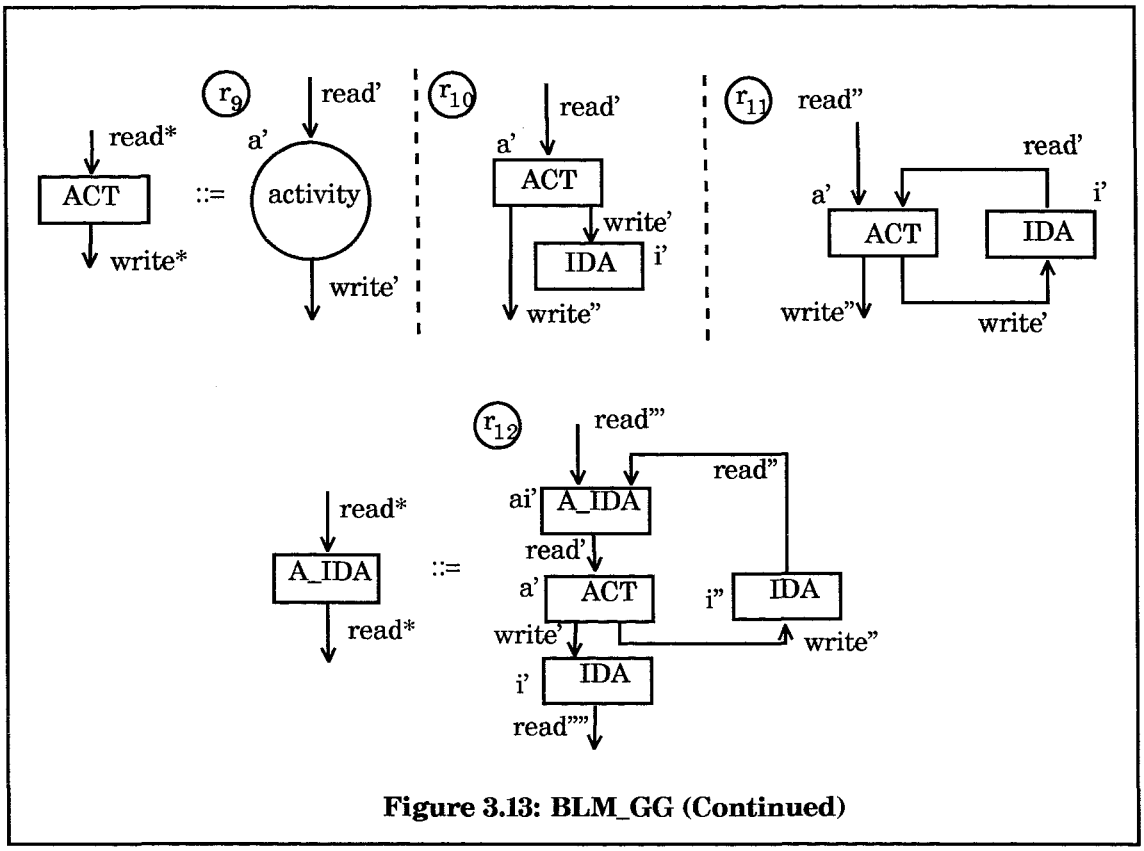


Figure 3.13: BLM_GG (Continued)

3.4.1. An Example Design

Figure 3.14 contains is a valid derivation of a MASCOT design using the BLM graph grammar, where $\Rightarrow^{x}$ indicates the re-writing of the non-terminal node on the left hand side with the daughter graph associated with the production rule x.

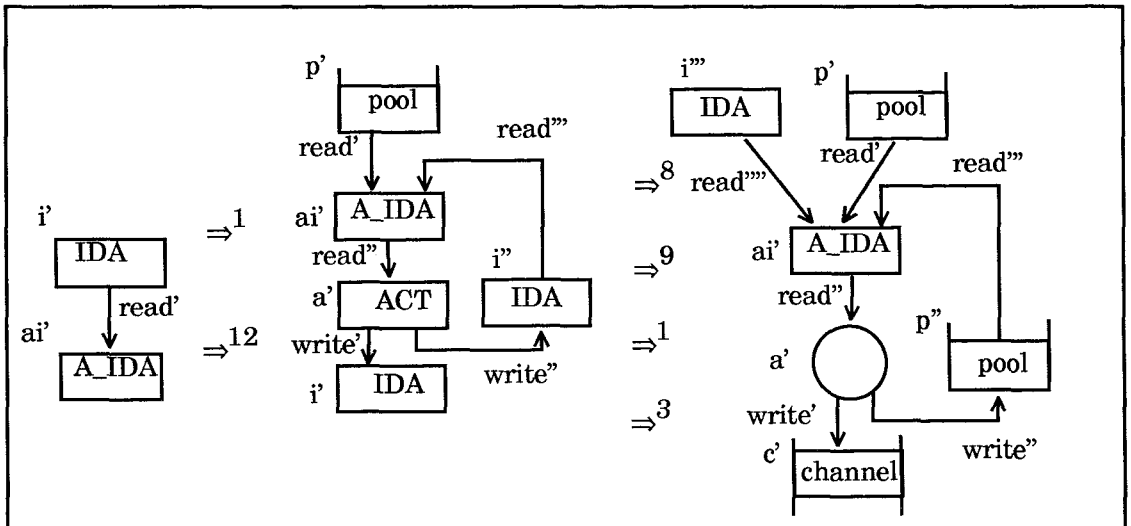


Figure 3.14: An Example BLM_GG Derivation

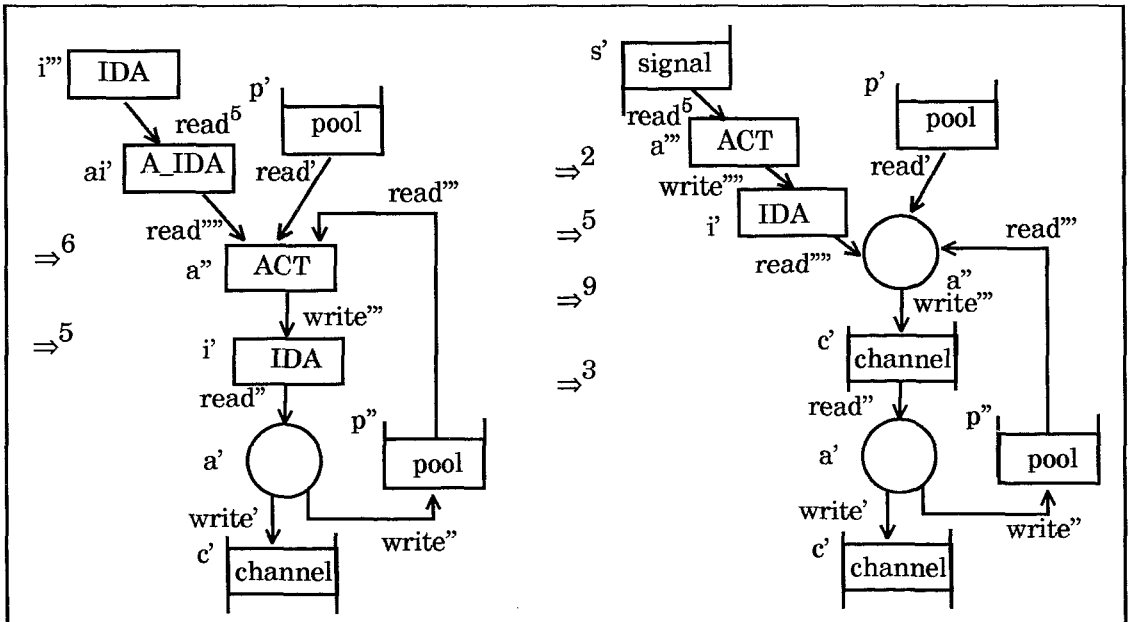


Figure 3.15: Example BLM_GG Derivation (Continued)

The derivation can be completed by applying rules r_9 and r_3 to the two nodes labelled with non-terminals, a''' and i' respectively.

3.5. Alternative Grammars for MASCOT

The twelve production rules for BLM designs are not the only rules which could be chosen to define the same class of designs. In particular, r_{12} could be replaced by r_{12}' given below. This has a simpler daughter graph, but it increases the ambiguity of the grammar.

r_{12}' = (ACT ::= ({a', ai'}, { (ai', a') }, {ACT, A_IDA}, {read'}, { (ai', A_IDA), (a', ACT) },
{ ((ai', a'), read') }),
{ (ai', read", channel, read*), (ai', read", pool, read*), (ai', read", signal, read*),
(ai', read", IDA, read*), (ai', read", A_IDA, read*) },
{ (a', write', channel, write*), (a', write', pool, write*), (a', write', signal, write*),
(a', write', IDA, write*), }).

Pictorially, r_{12}' is:

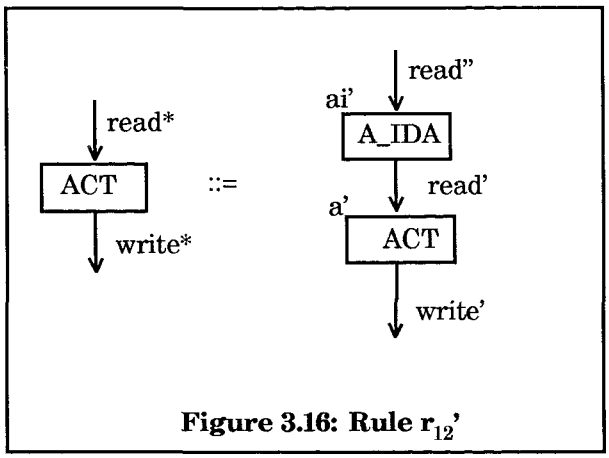


Figure 3.16: Rule r_{12}'

The extra simplicity of this rule is not judged to be sufficient to warrant the extra ambiguity it introduces.

Other grammars for the same class of MASCOT designs can be produced. For example, there is one which only contains the non-terminals ACT and IDA, and which only requires ten production rules (see Appendix 5). However, this is more ambiguous than BLM_GG. It also fails to include the SLM_GG production rules as a subset. For these reasons it is not considered further.

It is important to realise that rule r_6 of BLM_GG can be criticised for making the grammar ambiguous in sense that the same graph can have different derivations. For example, the design: [pool]→(activity)→[channel]→(activity)→[pool], can be derived using r1, r4, r5, r3, r9, r5, r9, r1 or r1, r5, r9, r6, r3, r5, r9, r1. Basically, either r6 or r4 can be used to unwrap the second activity and IDA. However, r_6 is not redundant, for without r_6

the number of MASCOT designs in the class L(BLM_GG) would be significantly reduced. For example, the back loops could only contain single IDAs and not arbitrary graphs.

This ambiguity of the grammar can be lessened, without changing L(BLM_GG), by removing r_6, and adding six extra production rules. These rules are needed to show that where an IDA labelled node occurs alone in a path in a daughter graph, an [IDA]→[A_IDA] sequence can also occur. Informally, such a grammar would contain the following extra production rules:

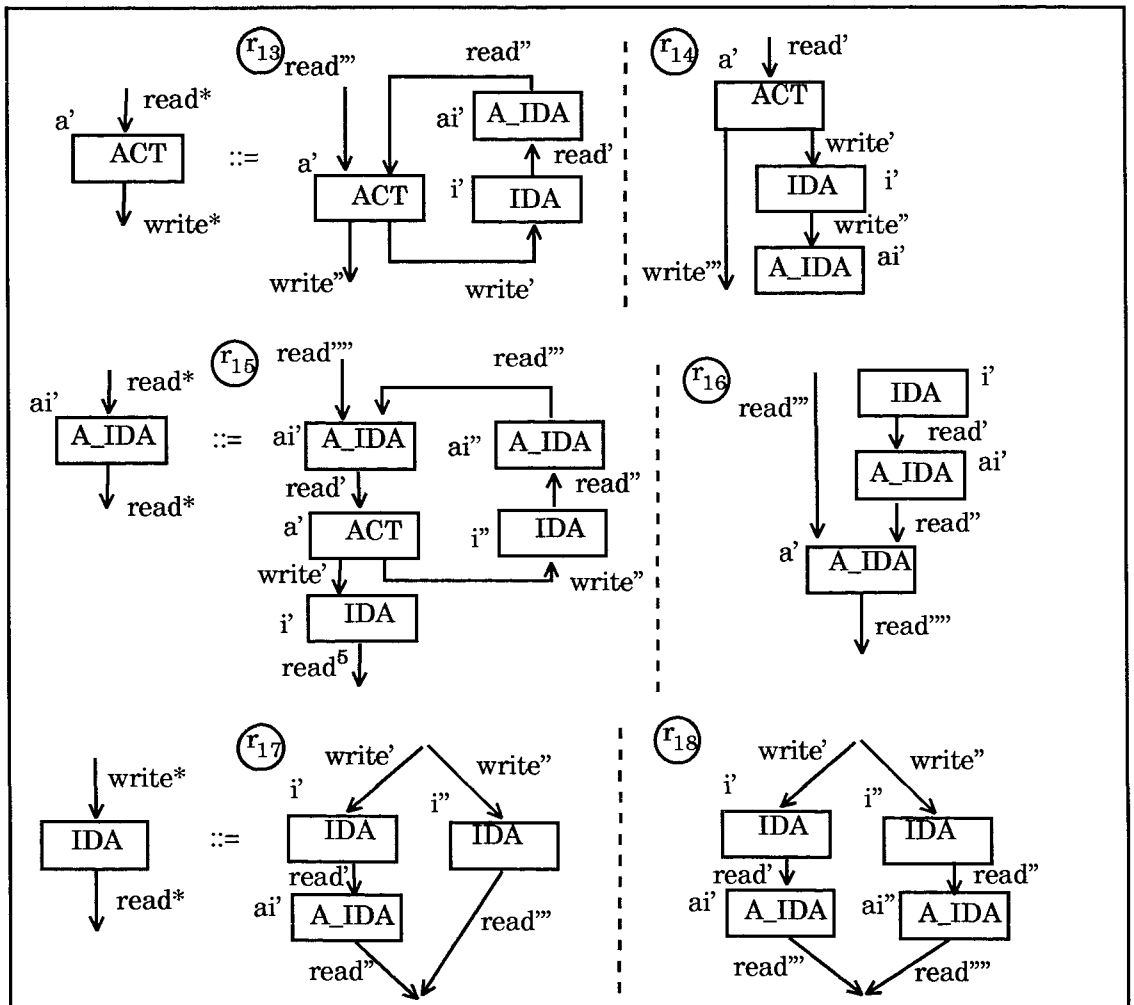


Figure 3.17: The Extra BLM_GG Rules

Given that BLM_GG is intended to define an abstract syntax of BLM designs, and so does not need to be unambiguous, this extra complexity is not needed. Also, unfortunately, this extended grammar is still slightly ambiguous, and so is still not suitable for a concrete

syntax of BLM designs. Consider, for example, the design in Figure 3.18. This can be generated by r4, r5, r10 and r5, r14.

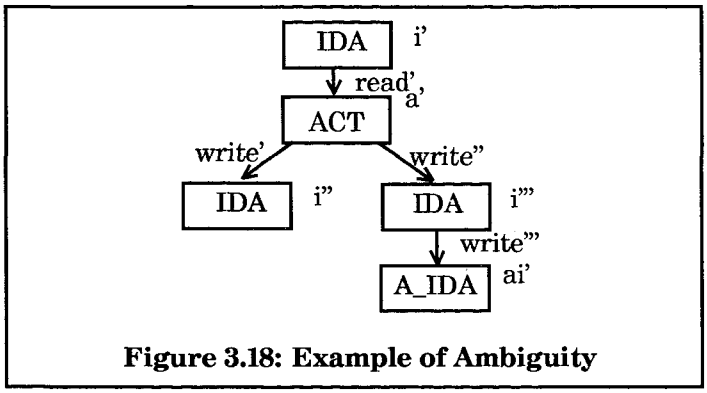


Figure 3.18: Example of Ambiguity

Due to its extra complexity, and because not all ambiguity is removed by exchanging rule 6 for rules 13 to 18, this extended grammar will not be considered further. Nevertheless, a less ambiguous grammar for BLM designs has been presented should one be required for other purposes.

Not all the designs which were described as being well-formed in chapter 2 are valid BLM designs. In particular, BLM designs must be connected graphs, pools can only have single readers, and channels and signals can only have single writers. None of these restrictions need hold for a well-formed MASCOT design. It is believed that a graph grammar could be defined to generate designs which were not constrained by these restrictions. However, it is further believed that it will contain more non-terminal symbols and significantly more production rules. It is not anticipated that there will be any problems with extending the denotational semantic definition to cover such an extended grammar.

Chapter 4: Semantic Models for MASCOT

This chapter explores possible formal semantics for MASCOT. It starts with a discussion of the features of MASCOT designs that will need to be captured by a suitable model. This is followed by a brief survey of existing formal languages, so that the promising ones can be identified and their semantic foundations examined further. Attention is then turned to the various options in developing a formal semantics for a concurrent network. The chapter concludes with a firm proposal for use of CSP in defining the formal foundations of MASCOT. This is worked out in detail in the next chapter. The aim of this chapter is to motivate the choice of CSP over the other formal models.

4.1. MASCOT Intuitions

As already discussed in chapter 2, MASCOT designs consist of two fundamental entities: activities and IDAs, which are linked by point-to-point data flows. An acceptable formal model for MASCOT needs to handle these entities naturally. It is important that a semantic model carries some intuitive conviction of correctness, as this is the best validation for it. A natural correspondence between the model and the design entities is a good way to achieve this. Furthermore, an acceptable semantics ought to enable a network to be reasoned about.

4.1.1. Activities

An acceptable semantic model of activities needs to capture the concept of a concurrent process. It is unlikely that semantic models of formal specification languages which do not have a process concept will be suitable for MASCOT designs. This assumption is considered further in the section 4.2, where languages are considered such as VVSL[24], that model concurrency without the process concept.

According to [JIM87], page 4-10, "The MASCOT definition does not prescribe the use of any particular scheduling strategy...". Further, a MASCOT design makes no commitment to either a single or multiple processor implementation, and with multi-processor implementations there is no commitment to either a single kernel, or separate kernels for each processor. This means that a formal semantics for MASCOT

[24]VIP VDM Specification Language, where VIP = VDM for Interfaces of the PCTE (Portable Common Tools Environment).

networks needs to reflect this ambiguity. Although reflecting this ambiguity reduces what may be deduced about the behaviour of a MASCOT design, the failure to do so would greatly limit the generality of the resulting formal semantics.

As explained in [Sim86], it is intentional in MASCOT that there is no explicit ordering of the activities. It can be argued that this ambiguity brings implementation freedom, while still conveying important information about the behaviour of the system through the dynamic protocols imposed on communications between activities by the IDAs. However, while the scheduling details may be left under-determined, there needs to be some basic fairness assumptions about activity progress and data propagation through the network. Unfortunately, these fairness assumptions remain unstudied in the MASCOT literature, certainly in anything approaching a formal framework. No position will be adopted concerning what fairness assumptions ought to be incorporated in a MASCOT semantics.

4.1.2. IDAs

IDAs are the data holding elements of a MASCOT design. In MASCOT-3, an IDA's protocol need not be specified, (such an IDA is known as a general IDA), but in the MASCOT subset formalised here, IDAs must possess one of the three kinds of protocol: i.e. pool, channel, or signal protocols. Any semantic model will have to be able to capture these kinds of protocol.

The IDA concept in MASCOT-3 includes a rich access interface notion; this allows for the type structure of the data handled by the IDA to be defined, along with procedures which may manipulate the data as it enters and leaves the IDA. The "windows" element of a MASCOT-3 IDA enables this access interface to be different to for each activity connected to it. However, the BLM concept of IDA is simpler, in that only implicit reading and writing procedures are associated with an IDA's data, and only one of these is visible to an activity per connection to the IDA[25]. The intended implicit semantics of the reading and writing procedures is that the data values are placed in the IDA, and taken from it, without being modified in any way. It can be argued that there a need in a model of MASCOT to capture the type structure of the

[25]The definition of BLM does not exclude the possibility of an activity being connected twice to the same IDA - see rule 11 of BLM_GG.

data that a given IDA can handle[26]. Signals and channels have a possibly zero place buffer associated with them. A semantic model of such IDAs has to capture the size of the buffer, and be able to record the passage of data through the buffer.

4.1.3. Paths

The point-to-point data flow of an MASCOT design obviously needs a semantic model. This does not mean, however, that an explicit point-to-point semantic model must be adopted, as broadcast communication models can usually be used to model point-to-point communication through the use of suitable conventions. Nevertheless the semantic model of a MASCOT design ought to be constructed in such a way that data can only be communicated between activities and IDAs which are connected. The data flow through the network must also obey the direction imposed by the read/write IDA interfaces.

4.2. Various Formal Languages

Numerous formal specification languages have been advocated at one time or another as suitable languages for specifying and describing systems. These languages may be broadly classified as: algebraic or axiomatic languages; model-based languages; languages with inter-conditions; modal or temporal logics; nets; and process algebras. A language is only "formal" if it has a mathematical semantics. Therefore, determining formal specification languages that are broadly suitable for describing MASCOT designs will automatically identify semantic models that may be adopted for MASCOT.

4.2.1. Sequential Languages

Algebraic and model-based languages such as OBJ, [GoT79], and VDM, [Jon90], may be discounted as they do not purport to support reasoning about concurrent systems, and in particular, provide no notation for defining the behaviour of separate processes.

[26] However, unfortunately, the work presented in this thesis does not include support for such type information.

4.2.2. Languages with Inter-conditions

Languages with inter-conditions are more interesting, as they tackle the problem of concurrent interactions. Examples of languages with inter-conditions include: Jones' extended VDM, [Jon83]; Stølen's work, [Stø92]; Xu and He's work, [XuH91] and [XuH92]; and Middelburg's VVSL, [Mid89a], [Mid89b] and [Mid92]. A language with inter-conditions is one where the state of the system, and the operations which manipulate that state are identified, and each operation is defined not only by its input / output relation, but also by a condition or conditions which hold while it is executing.

Jones added Rely and Guarantee conditions, [Jon83], to the usual pre and postconditions of VDM used to define an operation, [Jon90]. These are first-order predicates over the (shared) state, and assert invariants on the state that the operation can rely on holding while it is executing, and invariants that it must undertake not to violate. A specification consists of the definition of a shared state, and a set of such operations; the operations may execute whenever their preconditions hold. However, this approach fails to provide an explicit language for sequencing the operations, this has to be done in terms of state manipulation and precondition definition.

Stølen added a Wait condition, and Xu and He added a Run condition to Jones' basic rely and guarantee conditions. These assert (in first-order logic) that an operation may be blocked in certain states, and that an operation can rely on the environment establishing the Run condition if it is blocked, respectively. These extensions start to provide a language in which each operation may be considered to be a process that can deadlock; but it may be argued, the language is less expressive than the language of process algebras, introduced in section 4.2.5. In process algebras, the precise sequence of communications of a process can be spelt out, and deadlock can be deduced depending upon the sequence of communications the other processes can engage in.

The inter-condition in VVSL is different from the ones discussed so far, in that it is expressed in a form of linear temporal logic (LTL), rather than first order predicate logic. The use of LTL means that only one inter-condition is needed, instead of two or three. As will be reviewed in the next section, temporal logic has been advocated as a specification language for reactive systems; hence in VVSL each operation may be considered to be a concurrent reactive system. It can therefore be argued that in VVSL inter-conditions are powerful enough to turn operations into processes, and hence concluded that VVSL may be suitable for MASCOT. This is reinforced when it is

considered that IDAs may be seen as the shared states between the operation. However, unfortunately, the semantics of VVSL, given in [Mid89a], are very complicated, and currently do not have an associated proof theory. This mitigates against using this model for MASCOT. Another problem with VVSL for MASCOT, is that IDAs are more than just shared state, they also impose dynamic protocols. Now, while VVSL could emulate this effect by specifying it in the inter-conditions of the operations, this redistribution of important semantic content starts to make the mapping to MASCOT designs look unnatural.

4.2.3. Modal and Temporal Logics

Modal logics are logics in which the "mode" in which a given logical expression is true can be reasoned about, [HuC72]. Modal operators include: *Necessarily* and *Possibly* in Modal logic; *Must* and *May* in Deontic logic; *Know* and *Believe* in Epistemic logic; and *Always* and *Eventually* in Temporal logic (TL). It is usually a TL that is advocated for system specification, although deontic operators have been used in system specification logics such as Modal Action Logic, [GoF91]. Nevertheless, TL is the logic which has been more extensively advocated, and which will be considered further in this section.

The model chosen for a TL, that is, the structure that TL expressions are interpreted in, inevitably reflects a particular view of the nature of time, and this influences the expressive power of the logic. TLs with models that view time as a linear sequence are referred to as Linear Temporal logics (LTLs); TLs with models that take the view that there are different possible futures are referred to as Branching Temporal Logics (BTLs); and TLs with models which reflect that there can be more than one possible history which arrives at a point, as well as more than one possible future from a point, are referred to as Partial Order Temporal Logics (POTLs). Temporal logics can also be characterised by whether they view time as discrete or dense. Also, most temporal logics only enable the relative ordering of events to be reasoned about. However some logics have been extended with metrics which enable time to be handled quantitatively, eg. [Koy89]. In the rest of this thesis, TL will be used to refer to non-metric and discrete logics, but without commitment to either LTL, BTL or POTL.

Manna and Pnueli have advocated TLs as specification languages for reactive[27] systems on many occasions, [MaP81], [Pnu86a], [Pnu86b], and [MaP92]. TL expressions

[27] A definition and discussion of reactive systems can be found in Appendix Two.

can be used to assert the desired ordering of system events, including the required response to inputs. However, in spite of TL being a good specification language, since it does not presuppose a particular concurrent design for a given reactive system, it is not an ideal language for describing MASCOT designs for the same reason. A MASCOT design has particular activities, IDAs, and communication paths, and these have to be captured by a suitable semantics. Therefore TL semantics (usually the Kripke possible worlds model), are unlikely to be an ideal semantics for MASCOT.

4.2.4. Net Languages

Nets, or Petri-Nets, [Thi87] and [Rei87], are an extensively studied class of formalisms for modelling concurrent systems: they range from the simple Elementary Nets, [Thi87], with one token per condition, to infinite tokens per condition Place / Transition nets; [Rei87], to the relatively complicated, Predicate / Transition Nets, [Gen87] and Coloured Petri-Nets, [Jen87]. The basic idea of Petri-Nets is a net consisting of nodes and arcs, and the traversal of the net by tokens. In the different models tokens move around the net under different conditions, but in each model the movement of a token indicates some action, and the structure of the net some causal relation. Predicate / Transition nets allow tokens to be individualised, and logical guards to be associated with transitions. Coloured Petri-Nets associate information with the tokens, information that can be inspected and modified as the tokens move around the net.

Petri-Nets have been used to define the semantics of other design notations, for example, DeMarco Data Flow Diagrams, [TsP89]. Undoubtedly, Petri-Nets could also be used to describe MASCOT designs. The position of separate tokens in almost de-coupled nets could be used to represent the state of different activities. Communication between the activities using different IDAs would have to be explicitly modelled by the nets fragments that link the nets modelling the activities. The limited number of IDA protocols should mean that only a limited number of such linking net fragments need be developed. The need to model naturally the data holding nature of IDAs suggests that either Predicate / Transition or Coloured nets might be more appropriate than basic Place / Transition nets.

Petri-Nets are often given a semantics in terms of more fundamental models of concurrency, such as labelled transition systems or labelled asynchronous transition

systems, both of which are described later. These models are also used as semantics for Process Algebras, which, it will be argued, are also suitable for describing MASCOT designs.

4.2.5. Process Algebras

Process Algebras are based on the use of algebraic laws to describe the interactions of concurrent processes. Numerous process algebras have been advocated over the last decade or so, including: CCS (Calculus of Communicating Systems), [Mil89]; CSP, [Hoa85]; and ACP (Algebra of Communicating Processes), [BaW90a].

CSP is the notation which will be considered further as a representative of this class of formal language. CSP is chosen for a number of reasons, including: the fact that the more fundamental treatment of concurrency possible with CCS because of its τ-operator is not needed to define a semantic model for MASCOT; CSP is better suited as a "programming notation" for the designer to use to capture the behaviour of activities (as that is what it was initially intended for, [Hoa78]); and, most importantly, trace semantics are already defined for CSP. It is argued later that Hoare traces should form a suitable model for MASCOT.

4.2.5.a. Describing Activities

Processes are fundamental to process algebras, so their semantic models should have no trouble with handling the activities of MASCOT.

For example, CSP, [Hoa85] has three increasing expressive semantics, known as (Hoare) traces; failures; and failures+divergences, [BrR84]. Traces only capture the possible histories of the processes; there is no scheduling commitments, but nor are there any progress or fairness commitments. For example, the null trace is a possible behaviour of any process. The failures model enables the fact to be captured that at certain points in its history a process may refuse to engage in an action. While this helps in distinguishing non-deterministic programs, it still fails to require a process to progress. The addition of divergences, that is, histories after which the process behaves chaotically, also fails to require a process to progress. This is interesting as fairness properties are usually defined with respect to some progress requirements, [AFK87] and [Kwi89].

To conclude: CSP processes are a natural way to describe MASCOT activities, and CSP only has an informal progress assumption, as does MASCOT. The formalisation of this progress assumption, while increasing what may be deduced about a MASCOT design, would impose constraints that a MASCOT run-time system must satisfy. It can be argued that this is better done with a significantly richer formal model which takes into account the current limitations of scheduling theory, [ABR93].

4.2.5.b. Describing IDAs

The suitability of process algebras' semantic models may be questioned because of concerns over whether a process algebra can adequately describe IDAs. CSP, for example, is based on a synchronous message passing communication mechanism, while pool IDAs are shared variables, and it has been shown that shared variable communication may be implemented in a truly asynchronous way, [Sim90b], [Sim92] and [Ben92]. Further, MASCOT IDAs generally may be implemented as either passive or active components, while CSP only has a notation for active processes.

However, before process algebras are dismissed, the possibility of using a process to model an IDA needs to be considered. The objection that IDAs may (or indeed, will often) be implemented as a passive component is not relevant when it is realised that the semantic model only needs to capture its behaviour, and does not necessarily imply or constrain any particular implementation approach.

A model of an IDA based on a process is not completely un-intuitive when it is realised that the dynamic protocols associated with IDAs mean that a channel or a signal IDA can influence (via synchronisation when it is full or empty) the behaviour of the activities connected to it. An IDA may also interact with activities by alerting reading activities to the presence of new data to be read.

The main reason why it may be inappropriate to view an IDA as a process is that the accessing of an IDA via two or more different access interface windows may be implemented so that they can overlap in time. This is the fundamental idea behind the asynchronism of MASCOT designs, especially in distributed implementations where processors do not share a common clock. However, a CSP

process can only do one thing at a time; so it may be argued that a process model of an IDA is bound to impose some unnecessary synchronisation.

Nevertheless, on analysis, the strength of this argument relies on a reference to time. The assumption that a process cannot perform two actions at the same time is obviously correct for an implementation of a process, but it is less clear that it holds for mathematical model of a process which has no model of time associated with it. In fact, the basic model of a CSP process is a set of traces of actions that the process can engage in. The actions are atomic, and have no time lapse associated with them. Indeed when time is added to CSP, as in Timed CSP, [DaS89], the time lapse is associated with the transition between actions[28]. Therefore, in un-timed CSP, it is possible to interpret a process model of an IDA as engaging in more than one action simultaneously, albeit in sequence. The presence in the traces of the process of the actions in both orders is how in CSP the "concurrency" of the actions is modelled. Clearly, such a CSP model could give no indication as to how an asynchronous IDA may be implemented, but, as has been argued, this is not its task.

It may be considered perverse that CSP, a notation which is based on the concept of synchronous communication, is being investigated as a language for describing IDAs which can be used to support asynchronous communication between activities, especially when there are process algebras that support asynchronous communication directly, eg. [BKT84], and [JHH89]. However, there is a difference of definition in asynchronous communication between these algebras and that which MASCOT designs support. In the asynchronous process algebras asynchronous communication is defined in terms of infinite (unbounded) channels, while in MASCOT it is in terms of un-hindered access to a variable, a variable whose integrity can be maintained using a four-slot mechanism, [Sim90b], rather than by synchronous access using blocking. The buffers associated with channels in MASCOT designs are always of finite length: the fact that they are intended to be implemented requires this constraint. It can therefore be seen that the issues

[28]Of course, the enforced requirement of a time delay is enough to prevent a Timed CSP process from being a model of an IDA, as actions could not be viewed to overlap in time. However it is un-timed process algebras and not Timed CSP which is being investigated as a formal language for MASCOT designs. Should the semantic model of MASCOT need to be extended at any time to enable reasoning about the temporal behaviour of such networks, the semantic model would have to be reconsidered from the beginning as the enforced delay of Timed CSP is unacceptable.

addressed by asynchronous process algebras are not relevant to the problem of modelling MASCOT.

4.2.5.c. Describing Paths

Communication in CSP is via synchronisation through shared actions in the alphabets of two or more processes. Appropriate control of the alphabets of processes can ensure that only point-to-point communication occurs. CSP contains a channel mechanism to make achieving this relatively simple. CSP is hence suitable for describing the communications within a MASCOT design.

4.2.5.d. Summary

Process algebras are a natural approach for describing activities, and processes can be used to model IDAs providing that the instantaneous atomic nature of the actions that access them is insisted upon. This allows the interleaved model of concurrency to be used to capture the concurrency of actions. Thus the semantic models of process algebras, such as Hoare Traces, are likely to be suitable models for MASCOT designs. The following section explores the options that occur in choosing a formal model, to ensure that the choices made for process algebras, and CSP in particular, are suitable for MASCOT.

4.3. Semantic Model Options

In this section some of the numerous models of concurrency are compared. They are classified according to a taxonomy used by Winskel and Nielsen, [WiN92] and [Nie92]. This distinguishes models depending upon whether or not concurrency is modelled as non-determinism; on whether or not the model has a state element; and on whether the history is branching or linear. This taxonomy is not completely general, as some models of concurrency do not fit into it neatly. Nevertheless, it is used as it helps to highlight various semantic options that must be made in developing a semantic model for a concurrent system.

The following models are discussed during the presentation of the taxonomy of modelling options: Labelled Asynchronous Transition Systems, (LATS); Labelled Event Structures, (LES); Mazurkiewicz Traces, (MT); Labelled Transition Systems, (LTS);

Synchronisation Trees, (ST); and Hoare Traces (HT). Their definitions can be found in Appendix Three.

Category theory, [BaW90b], has been used to formalise the relationship between these various models, [Win85] and [WiN92]. However, the various models are only compared informally here.

4.3.1. The Interleaved - Non-Interleaved Option

The various models of concurrency can be partitioned into those which model concurrency in terms of non-determinism, and those that do not. Those that do, model the concurrency of two events by enabling them to be arbitrarily interleaved, while those that do not usually introduce a notion of causality into the model. An action which is caused by another action cannot happen concurrently with that action, while actions which are not causally related are free to occur concurrently.

Labelled Asynchronous Transition Systems, (LATS); Labelled Event Structures, (LES); and Mazurkiewicz Traces, (MT), are all examples of models which take the non-interleaved option. These models are also known as partial order models, in that only causally related actions are ordered, while the order of occurrence of non causally related actions cannot be determined. This is sometimes referred to as a *true* concurrency model, and is suitable as a model of a distributed system, where there is no synchronisation of events on different machines to enable an ordering to be established between them.

4.3.2. The State - Behaviour History Option

The various models of concurrency can also be partitioned into those that capture the information about the states of the system, and those that only record the history of the observable actions that the system may engage in. Clearly, a finite, cyclic, state machine may produce infinite behavioural histories; therefore the state based models are in many ways more compact than their infinite behavioural histories. Nevertheless, history models are often preferred because they are more abstract; they do not record information which cannot be perceived by an outside observer of the system.

Labelled Asynchronous Transition Systems (LATS), and Labelled Transition Systems (LTS), are examples of models which capture the state information. The other models under consideration do not.

4.3.3. The Branching - Linear Option

The models of concurrency which adopt the behavioural history approach can be partitioned into those that record the branching structure of the behaviour and those that only record the linear histories of the system. The branching models have the advantage of being able to distinguish between the occasions when a particular non-deterministic choice is made in a system's history. Linear models are more abstract, but cannot distinguish certain systems.

Labelled Event Structures (LES) and Synchronisation Trees (ST) are examples of branching history models, while Mazurkiewicz Traces (MT) and Hoare Traces (HT) are examples of linear history models.

This map of the various models is presented in Figure 4.1.

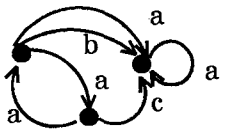
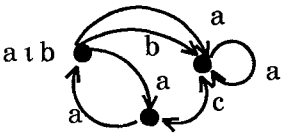
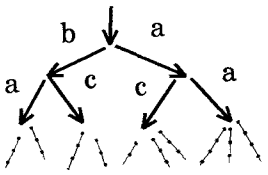
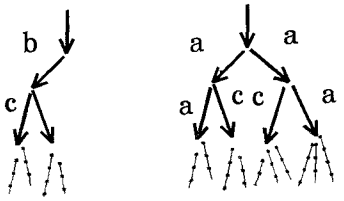
	Interleaved	Non-Interleaved
States	Labelled Transition Systems 	Labelled Asynchronous Transition Systems 
Branching Behaviour	Synchronisation Trees 	Labelled Event Structures 
Linear Behaviour	Hoare Traces { {}, a, b, ab, ba, .., }	Mazurkiewicz Traces { a, ac, aa, aac, aaa, ... b, bc, bcb, ... }

Figure 4.1: A Map of the Semantic Options

The position of each model in the above map correlates to the distinguishing power of that model, as shown in Figure 4.2.

Pairs of Processes	LTS	ST	HT	LATS	LES	MT
$a \rightarrow P \mid\mid b \rightarrow Q$ $a \rightarrow P \mid b \rightarrow Q$	✗	✗	✗	✓	✓	✓
$a \rightarrow (b \rightarrow P \mid c \rightarrow Q)$ $(a \rightarrow b \rightarrow P) \mid (a \rightarrow c \rightarrow Q)$	✓	✓	✗	✓	✓	✗
$\mu X : \{a\} . (a \rightarrow X)$ $a \rightarrow a \rightarrow a \rightarrow a \rightarrow a \rightarrow a \ldots$	✓	✗	✗	✓	✗	✗

Figure 4.2: The Distinguishing Power of the Models

4.4. Semantic Options and MASCOT

There is a strong case for arguing that a natural model for MASCOT will be behavioural rather than state-based, namely, because state is not a prominent concept in MASCOT designs. However, there are no clear grounds for deciding between linear and branching behavioural models: it depends upon the detail that the design needs to be modelled. Also a MASCOT design may be given an interleaved or a non-interleaved semantics. There is nothing in MASCOT which makes the interleaved abstraction particularly inappropriate, although if action refinement was required, for example, to allow MASCOT designs to be verified against significantly more abstract specifications, non-interleaved models are required, [Ace92].

4.5. Conclusion: A Proposed Model

The above discussion of formal semantics for concurrent systems has indicated that HTs are the most abstract model which has been considered. Winskel and Neilsen have formalised this informal intuition using category theory, [WiN92] and [Nie92]. There are strong advantages in adopting abstract semantics over more concrete semantics. In particular, abstract semantics tend to be the simplest, and hence the most comprehensible; and the easiest to reason about. However, the disadvantages of abstract semantics are that: some intuitions developed from the real phenomena being modelled may not hold; certain properties may not be modelled, and hence other properties may not be able to be demonstrated to hold. For example, liveness properties cannot be demonstrated for systems with HT semantics. The position adopted here, as in [Moo90], is that the advantages of HTs are such that the disadvantages can be tolerated. Clearly, it would be desirable to be able to reason about the liveness properties of a MASCOT design, but it is a reasonable initial goal to investigate a semantics for which enables safety properties of MASCOT designs to be demonstrated.

HTs, as well as being an abstract model of concurrency, have also been used a semantics for CSP, a process algebra. Furthermore, it has been argued that process algebras are languages which allow MASCOT designs to be described relatively naturally. Therefore CSP with HT semantics will be used as the basis for a language for describing MASCOT designs in the following chapter.

Chapter 5: MASCOT Communicating Activities

This chapter contains a definition of a denotational semantics for MASCOT designs. The definition is structured using the the Branching / Looping MASCOT graph grammar (BLM_GG) presented in chapter 3. This is interpreted as defining the abstract syntax of MASCOT designs. The semantic model used is Hoare Traces, and it is defined by describing MASCOT designs as networks of CSP processes, [Hoa85].

Once the semantic model is defined, its utility is demonstrated with a small example of how it facilitates reasoning about MASCOT designs. In particular, it is shown how safety properties may be proven with respect to predicate-over-traces specifications, and deadlock freeness may be demonstrated.

5.1. A Denotational Semantics for MASCOT Designs

The BLM graph grammar defined in section 3.4, is used in this section to present a denotational semantics of MASCOT designs. The semantics are expressed in terms of CSP processes. The primary advantage of defining the model via CSP rather than directly is that the well-formedness of the semantic domain has already been demonstrated. In particular, recursion in CSP has been shown to have a fixed-point semantics, [Hoa85].

In this section, the function which maps from MASCOT to CSP will be denoted by M. M is sometimes called the valuation function, [Sch86].

The daughter graphs on the right hand side of the productions of the BLM_GG will be mapped into parallel compositions of CSP processes, with suitable re-labelling to ensure communications and suitably disjoint alphabets. The nodes labelled with terminal symbols of the grammar (the IDAs and activities of the design) will be mapped directly into CSP processes. The meaning of their composition in the design is given by the model assigned by M to the production rules which were used in the design's derivation from the initial graph of the grammar. The initial graph of the grammar is also given a semantic model by M.

As explained before, BLM_GG is the abstract syntax of MASCOT, so the denotational semantics given by M is calculated for a given design *and* a given derivation. The ambiguity of BLM_GG therefore does not introduce problems by defining more than one different semantics to a design.

The essence of this model of MASCOT is that a MASCOT design is a network of CSP processes. IDAs map into fixed CSP processes, defined below, while activities are user defined CSP processes. A MASCOT design is therefore modelled as a CSP design where general processes may only communicate with each other via one of the three pre-defined processes.

5.1.1. CSP Syntactic Conventions

Designers should use the CSP notation which is proposed in this section when defining the behaviour of MASCOT activities in formalised MASCOT designs. It is a slightly extended notation from that found in [Hoa85]. Extensions are defined which make the type of the IDA which is being communicated with explicit in the text of the activity description. An important side benefit of using these notations and conventions is that actions in the activities and IDAs associated with communication automatically synchronise without the use of re-labelling functions in the semantic definitions.

Actions which model communication events are to have the following form:

VW(X.Y, Z)

where: V ∈ { c, p, s }; W ∈ { ?, ! }; X is the name of the IDA adjacent to the activity being communicated with; Y is the name of the link to X; and Z is the value communicated. V makes plain in the text of the activity the dynamic protocol associated with this communication link, where "c" stands for channels; "p" for pools; and "s" for signals. W indicates the direction of the flow of data: "?" for input to the activity, and "!" for output from the activity. For example: "P ≡ p?(i1.left, x) → P(x)" describes an activity which repeatedly reads from IDA i1 on link "left", where i1 has a pool protocol.

Functions are defined for extracting the different components from these actions, similarly to the way in which channels are handled in CSP, [Hoa85]. Protocol(VW(X.Y, Z)) = V; Channel(VW(X.Y, Z)) = X.Y; and Message(VW(X.Y, Z)) = Z.

V!(X.Y, v) → P reduces to the CSP expression X.Y.v → P and V?(X.Y, x) → P(x), reduces to y:{y | Channel(y)=X.Y)} → P(Message(y)). Hence, these extensions reduce to the normal conventions used for CSP channels in [Hoa85].

A function, called Links, is assumed to exist, which takes a process and returns the set of communication actions in which the process can engage. For example,

Links(μP: {i2.right.x, a} ∪ {a | Channel(a)=i1.left}.

(p?(i1.left,x) → a → c!(i2.right,x) → P))

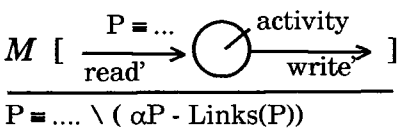
returns the set { i2.right.x } ∪ {a | Channel(a)=i1.left}

5.1.2. Semantics of Activities

Rule 9 contains a daughter graph with a single node, labelled with the terminal activity symbol. In a complete formal design, such a node would also be labelled with the CSP expression which defines the activity. Such an expression is needed to provide the functional definition of an activity; information which MASCOT, along with most design methods, fails to provide. A MASCOT design with this added functionality definition will be called a *Formal MASCOT design.*

The process expressions should make use of the conventions defined above for defining communication actions. P is this expression in the example given below. The valuation function, $\boldsymbol{M}$, defines the meaning of an activity to be the CSP process defined by the expression P, with all non-communication actions hidden. The hiding of these actions ensures that interactions due to common actions cannot occur in formal MASCOT designs.

Rule r_9



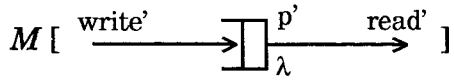
Links(P), the set of actions that are not hidden in the semantic definition of an activity, is the set union of the alphabets of the channels on which that activity communicates.

5.1.3. Semantics of IDAs

Rules 1, 2, and 3 have daughter graphs with a single node, labelled with the terminal pool, signal, and channel labels respectively. The valuation function, $\boldsymbol{M}$,

defines the meaning of these IDAs by mapping these daughter graphs to specific CSP processes. These processes model the protocols that these IDAs impose.

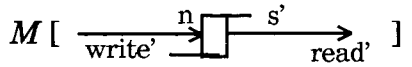
Rule r_1



p':μInit : {write'.x, read'.λ} . (write'?x → P(x) | read'!λ → Init)
p':μP(x) : {write'.x, read'.x} . (write'?x → P(x) | read'!x → P(x))

Init and P are the recursive CSP processes which define the behaviour of a pool IDA. Each instance of such an IDA is labelled with a unique name, denoted by p' in this example. Pool nodes are also labelled with initial value, λ. The labelling in the semantic model of the processes with the IDA's name, p', means that all the actions of the IDA are prefixed with "p'.". This is used to ensure that IDAs have disjoint alphabets, and so do not cause unwanted synchronisations. IDAs are defined using the standard CSP channel communication actions. After labelling, these actions correspond to the communication actions of activities.

Rule r_2

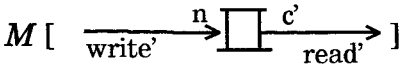


$n > 0$

$s':S^n_{<>} \equiv (write'?x \rightarrow S^n_{<x>})$
$s':S^n_{s^\wedge<x>} \equiv (write'?y \rightarrow S^n_{<y>^\wedge s^\wedge<x>} \mid read'!x \rightarrow S^n_{s})$ $\quad \#s < n$
$s':S^n_{s^\wedge<x>} \equiv (write'?y \rightarrow S^n_{<y>^\wedge s} \mid read'!x \rightarrow S^n_{s})$ $\quad \#s = n - 1$

S is a set of mutually recursive CSP processes which defines the behaviour of a signal IDA. Each instance of such an IDA is labelled with a unique name, denoted by s' in this example. A signal IDA is also labelled with the size of its buffer, n. The model is only defined for $n > 0$ and not $n = 0$, although strictly this is a possible signal protocol. The restriction is adopted to keep the model simple, but, if required, the model could be adapted to handle this case. However, the $n = 0$ case also requires modifications to the semantics of composite MASCOT designs, given below. In particular, actions will need to be re-labelled to ensure activities on both sides of such an IDA synchronise on communication.

Rule r_3



$n > 0$

$c':C^{n}_{<>} = (write'?x \rightarrow C^{n}_{<x>})$

$c':C^{n}_{s^{\wedge}<x>} = (write'?y \rightarrow C^{n}_{<y>^{\wedge}s^{\wedge}<x>} \mid read'!x \rightarrow C^{n}_{s})$ $\quad \#s < n$

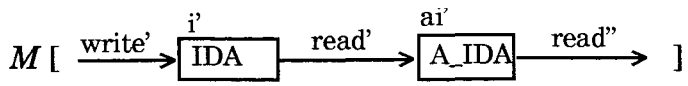
$c':C^{n}_{s^{\wedge}<x>} = (read'!x \rightarrow C^{n}_{s})$ $\quad \#s = n - 1$

C is a set of mutually recursive CSP processes which define the behaviour of a channel IDA. Each instance of such an IDA is labelled with a unique name, denoted by c' in this example. A channel IDA is also labelled with the size of its buffer, n. As for signal IDAs, only the n > 0 case is defined. The n=0 case is not considered for the same reasons as before.

5.1.4. Composite Semantics

In this section a denotational semantics for activity and IDA compositions are defined. The definitions are defined using rules 4 to 12 of the BLM_GG. First the meaning of the initial graph, Z, is given.

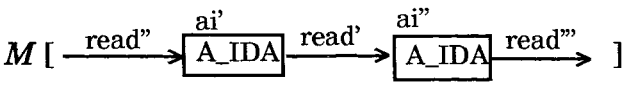
Initial Graph Z



(M[IDA] || M[A_IDA]) \ {c | c = *read'.*}

The meaning of this daughter graph, is the parallel composition of the meanings of the nodes. The naming conventions adopted for communication actions in activities and IDAs means that the actions communicating over the read' channel will automatically synchronise. The set {c | c = *read'.*} indicates that all the actions which communicate over this channel are hidden, and only the input and output actions (ie the actions on channels write' and read") of the overall system are visible.

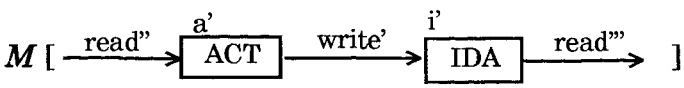
Rule r_4



(*M*[A_IDA]) || M[A_IDA]) \ {c | c = *read'.*}

The meaning of this daughter graph, is the parallel composition of the meanings of the nodes. Again the communications over read' are hidden.

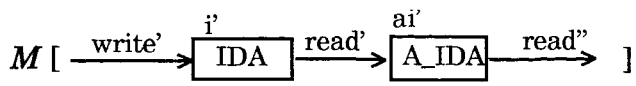
Rule r_5



(*M*[ACT] || M[IDA]) \ {c | c = *write'.*}

The meaning of this daughter graph, is the parallel composition of the meanings of the nodes. The actions which communicate over the write' channel are hidden.

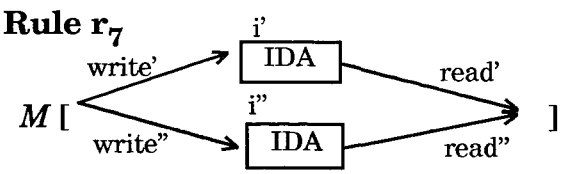
Rule r_6



(*M*[IDA] || M[A_IDA]) \ {c | c = *read'.*}

The meaning of this daughter graph, is the parallel composition of the meanings of the nodes. The actions which communicate on the internal channel are hidden.

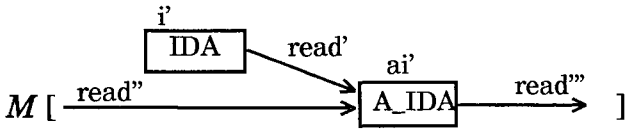
Rule r_7



(*M*[IDA] || M[IDA])

The meaning of this daughter graph, is the parallel composition of the meanings of the nodes. No actions are hidden, and no actions should synchronise; the alphabets of each IDA will be disjoint.

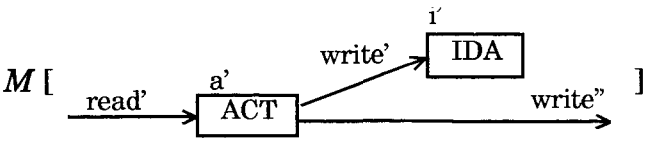
Rule r_8



(*M*[IDA] || M[A_IDA]) \ {c | c = *read'.*}

The meaning of this daughter graph, is the parallel composition of the meanings of the nodes. The actions which communicate on the internal channel are hidden.

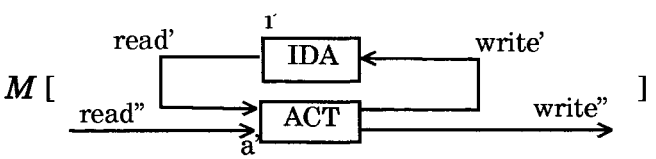
Rule r_{10}



(*M*[ACT] || M[IDA]) \ {c | c = *write'.*}

The meaning of this daughter graph, is the parallel composition of the meanings of the nodes. The actions which communicate on the internal channel are hidden.

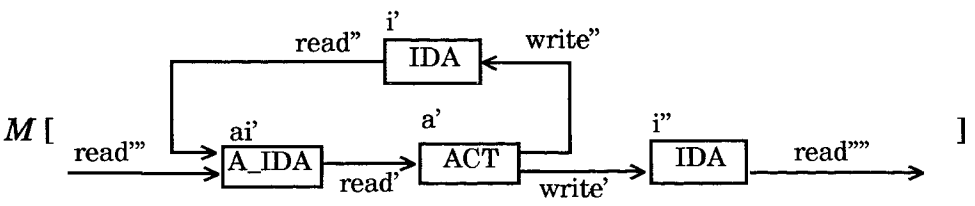
Rule r_{11}



(*M*[ACT] || M[IDA]) \ {c | c = *read'.* ∨ c = *write'.*}

The meaning of this daughter graph, is the parallel composition of the meanings of the nodes. The actions which communicate on both read' and write' channels are hidden.

Rule r_{12}



(*M*[A_IDA] || *M*[ACT] || *M*[IDA] || M[IDA])
\ { c | c = *read'.* ∨ c = *read".* ∨ c = *write'.* ∨ c = *write".* }

The meaning of this daughter graph, is the parallel composition of the meanings of the nodes. Actions which communicate on all internal channels are hidden.

5.1.5. Well-Formedness Rules for Formal MASCOT Designs

The rules presented in this section help to ensure that well-formed formal MASCOT designs descriptions are internally consistent.

Rule 1: A CSP process expression used to define an activity may not refer to an IDA to which it is not directly connected in the design graph.

Rule 2: A CSP process expression used to define an activity must refer to an IDA to which it is directly connected as defined in the design graph, and it must pass data to the IDA in the direction specified.

Rule 3: A CSP process expression used to define an activity may not refer to a process which is used in the definition of another activity.

Rule 4: A CSP process expression used to define an activity may not include the parallel, || , or hiding, \, operators.

Rule 5: All IDAs and activities must have unique names.

5.1.6. Traces

This section defines how the Hoare Trace model is calculated for CSP expressions. The definitions are repeated from [Hoa85]. They are included for completeness.

$$traces(\ STOP\ ) = \{\ \langle\rangle\ \}$$

$$traces(\ x{:}B \rightarrow P(x)\ ) = \{\ t \mid t = \langle\rangle \vee (t_0 \in B \wedge t' \in traces(P(t_0)))\ \}$$

$$traces(\ \mu X{:}A.F(X)\ ) = \bigcup_{n \geq 0} traces(\ F^n(\ STOP_A\ )\ )$$

$$traces(\ P/s\ ) = \{\ t \mid s \hat{}\ t \in traces(P)\ \}$$

$$traces(\ P \backslash C\ ) = \{\ t \lceil (\alpha P - C) \mid t \in traces(P)\ \}$$

$$traces(\ f(P)\ ) = \{\ f^*(s) \mid s \in traces(P)\ \}$$

$$traces(\ P \parallel Q\ ) = \{\ t \mid (t \lceil \alpha P) \in traces(P) \wedge (t \lceil \alpha Q) \in traces(Q)\ t \in (\alpha P \cup \alpha Q)^*\ \}$$

$$traces(\ P \sqcap Q\ ) = traces(P) \cup traces(Q)$$

These are the basic CSP expressions, a richer collection can be found in [Hoa85].

5.2. Reasoning About Formal MASCOT Designs

The motivation for formalising the *way of modelling* of MASCOT was to gain a means of reasoning about MASCOT designs. MASCOT's semantic definition in terms of CSP processes means that there are three kinds of formal reasoning about MASCOT designs that may now be carried out, namely: proof of equivalence; proof of safe behaviour with respect to a specification; and proof of absence of deadlock.

CSP's specification language is Predicates-over-Traces, and such specifications can now be used in specifying MASCOT designs. The Predicates-over-traces specification language consists of the use of first order logic with a library of functions for manipulating sequences of actions (traces) defined in [Hoa85]. Through being given a semantics in CSP, MASCOT designs can be proven to be safe with respect to such specifications using the SAT logic described in the subsection below. That is, it can be proved that if the MASCOT design does anything, it will do what has been specified. It cannot be proved that the MASCOT design will do anything, liveness properties cannot be demonstrated. An example of a safe

behaviour proof of a MASCOT design is given in the second subsection. The third subsection discusses the proof of deadlock freeness of MASCOT designs.

No algebraic laws are given for manipulating MASCOT designs, despite this being normal in the literature on process algebras. The reason for this is not that such laws could not be developed, but that their use is not advocated in this thesis. A MASCOT design is expected to be derived from the use of the MASCOT design method and philosophy. It is not expected that during normal system development such designs will be transformed into logically equivalent designs, through the use of algebraic laws. However, as the semantic model is expressed in CSP, at the semantic level MASCOT inherits the laws of CSP should they be needed.

5.2.1. The SAT Logic

The basic inference rules of the CSP sat logic are repeated in this section from [Hoa85].

L1 $$\frac{}{P \text{ sat } true}$$

L2A $$\frac{P \text{ sat } S,\ P \text{ sat } T}{P \text{ sat } S \wedge T}$$

L2 $$\frac{\forall n \bullet P \text{ sat } S(n)}{P \text{ sat } \forall n \bullet S(n)}$$

L3 $$\frac{P \text{ sat } S,\ S \Rightarrow T}{P \text{ sat } T}$$

L4 $$\frac{\forall x \in B \bullet P(x) \text{ sat } S(tr, x)}{(x{:}B \rightarrow P(x)) \text{ sat } (tr{=}\langle\rangle \vee (tr_0 \in B \wedge S(tr', tr_0)))}$$

L5 $\quad \dfrac{P \text{ sat } S(tr),\ s \in traces(P)}{(P/s) \text{ sat } S(s^{\wedge}tr)}$

L6 $\quad \dfrac{F(X) \text{ is guarded, STOP sat } S,\ ((X \text{ sat } S) \Rightarrow (F(X) \text{ sat } S))}{\mu X.F(X) \text{ sat } S}$

L7 $\quad \dfrac{P \text{ sat } S(tr),\ Q \text{ sat } T(tr)}{P \ \| \ Q \text{ sat } (S(tr\lceil \alpha P) \wedge T(tr\lceil \alpha Q))}$

L8 $\quad \dfrac{P \text{ sat } S(l, r),\ Q \text{ sat } T(l, r)\ \ P \text{ is left guarded or } Q \text{ is right guarded}}{P \gg Q \text{ sat } \exists s \bullet S(l, s) \wedge T(s, r)}$

An alternative version of L8 which does not use the pipe connective is given below.

L8' $\quad \dfrac{P \text{ sat } S(l, r),\ Q \text{ sat } T(l, r)\ \ P \text{ is left guarded or } Q \text{ is right guarded}}{(P \ \| \ Q) \setminus \{ c \mid c \in \alpha Shared\_Channel \} \text{ sat } \exists s \bullet S(l, s) \wedge T(s, r)}$

The requirement for P to be left-guarded or Q to be right guarded in L8 is to guard against livelock, which is where $P \gg Q$ spend all their time engaged in internal communication with hidden actions. A process, P, is said to be left guarded if $\exists f \bullet P$ sat $(\#r \leq f(l))$. This insures that there is a defined bound on the number of hidden communications that P will engage in on the r channel before engaging in a visible communication on the l channel.

It is worth considering whether there are any further rules which may be added to the SAT logic which are specific to CSP programs derived from MASCOT designs, given that such CSP will be of a stylised form.

One indication that there will not be any new rules is that the CSP which results from MASCOT designs is still very general. In particular, the CSP for activities is only slightly restricted (parallel and hiding operators are not allowed), and the CSP used to define IDAs is also quite rich. For example, pool IDAs cannot be specified by separate predicates on the traces of its channels. This is because the overall behaviour of a pool is dependent upon the relative ordering of the occurrences of reads and writes to the pool. The same is true of signals, but not channels. Hence, the rule L8 above is not generally applicable to MASCOT designs.

Any rules which would be applicable to MASCOT designs would also be applicable to a large subset of ordinary CSP. Also there are ordinary CSP rules which are already more specific than MASCOT can use. It is hence reasonable to suppose that such general laws would have been developed independently, given the vigour of research in the process algebra community.

5.2.2. An Example

A predicate-over-traces specification is: c".read" $\leq^3$ double*(c'.write'), where this is an abbreviation for message*(tr⌈c".read") $\leq^3$ double*(message*(tr⌈c'.write')), and f*(t) is the function which applies the function f to each member of the trace t. Double is the function which returns twice the value of its parameter. This specifies a design which accepts inputs along c".write', and which produces outputs on c".read" which are double the value of the inputs, and which may lag up to three values behind the inputs.

A possible MASCOT design consists of a single simple activity, and two interface IDAs with channel protocols. The design is the simplest well-formed MASCOT design: it was chosen to illustrate the reasoning which can be performed about formal MASCOT designs.

A derivation-tree of the design is given in Figure 5.1.

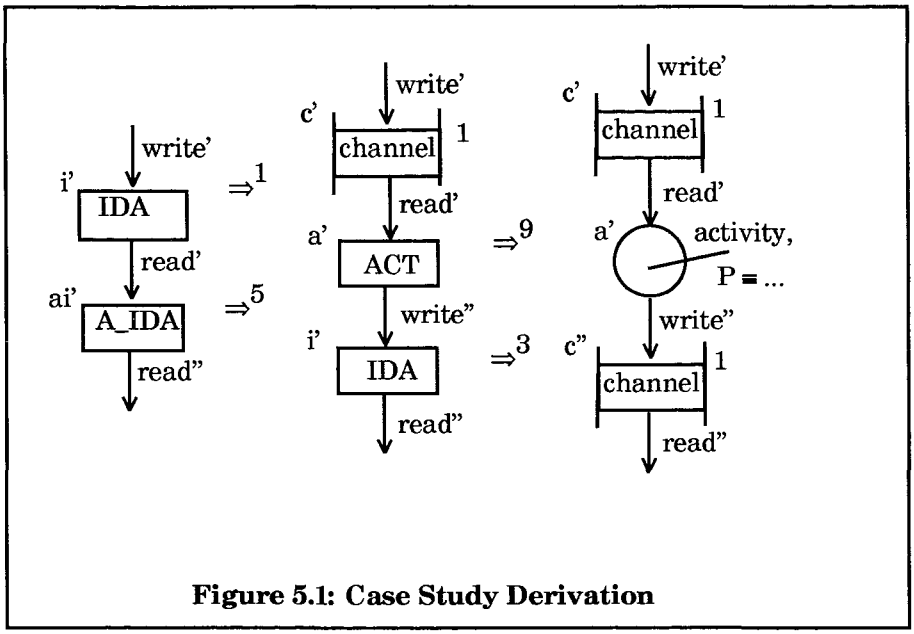


Figure 5.1: Case Study Derivation

The semantics of the design, calculated using the rules given in section 5.1, is:

(*M*[IDA] || *M*[A_IDA]) \ {c | c = *read'.*} ***Initial Graph***

(c' || (*M*[ACT] || M[IDA]) \ {c | c = *write".*}) \ {c | c = *read'.*}

Rules r1, r5

(c' || (P = ... \ (αP - Links(P)) || c") \ {c | c = *write".*}) \ {c | c = *read'.*}

Rules r9, r3

If the alphabets of the three CSP processes are disjoint, except for communications over channels, the model can be tidied up to become:

(c' || P || c") \ {c | c = *write".* ∨ c = *read'.* ∨ c ∈ αP - Links(P) }

The functionality chosen for P will be simple: the consumption of data from the input IDA, and the production of data at the same rate, only at twice the magnitude. That is, P = c?(c'.read', x) → c!(c".write", 2*x) → P.

Links(P) is the same as αP, resulting in the further simplification of the model.

(c' || P || c") \ {c | c = *write".* ∨ c = *read'.* }

The proof of the correctness of the design is given in Figure 5.2, below. The proof is presented in a natural deduction style, but it is only discharged rigorously.

5.2.3. Reasoning About Deadlock

The proof obligation which needs to be discharged to show that a given MASCOT design, D, is free of deadlock is:

$\forall s \in traces(D) \bullet (D/s) \neq STOP$

The way to discharge the proof obligation is to consider the various states a design could be in after an arbitrary trace, s, and to demonstrate that for each state the design may engage in an action.

Consider, for example, the design used above. After an arbitrary trace, either the activity will be ready to read from c' or it will be ready to write to c". If the former, c' may be either full or empty (the buffer is only 1-place). If empty, the system is ready to accept an input from the environment as soon as it is provided, so it is not in deadlock. If full, the activity may read the value in c', so again the system is not deadlocked. If the activity is ready to write to c", c" may be either empty or full. If empty, the activity may proceed to write its value, so again the system is not deadlocked, while if c" is full, the system is ready to have the value read by the environment, so again the system is not deadlocked. The design is therefore deadlock free.

It is, of course, not surprising that this example is deadlock free. Deadlock only occurs in a design where pairs of processes only communicate on single channels, and there are no loops in the design, when a process stops. Nevertheless, the argument about deadlock freeness of the design does demonstrate the kind of deadlock reasoning that may be carried out about MASCOT designs, now that they have a CSP semantics.

There is work on the development of automatic algorithms which can determine whether a CSP program is deadlock free, [BrJ92], [FS93]. The essence of these approaches is one of model-checking, the examination of an exhaustive model constructed from the CSP description.

from c', P, c"
1: from P = μX. (c?(c'.read', x) → c!(c".write", 2*x) → X)
 1.1: STOP sat (tr=<>) -- L4
 1.2: tr=<> ⇒ c".write" $\leq^1$ double*(c'.read') -- Logic, $\leq^n$
 1.3: STOP sat c".write" $\leq^1$ double*(c'.read') -- L3, 1.1, 1.2
 1.4: from X sat c".write" $\leq^1$ double*(c'.read')
  1.4.1: c?(c'.read', x) → c!(c".write", 2x) → X
   sat (tr ≤ <c'.read'.x, c".write".2x> ∨
   (tr ≥ <c'.read'.x, c".write".2x> ∧
   tl(c".write") $\leq^1$ tl(double*(c'.read')))) -- L4, 1.4
  1.4.2: tr ≤ <c'.read'.x, c".write".2x> ⇒
   c".write" $\leq^1$ double*(c'.read') -- Logic, ≤
  1.4.3: tr ≥ <c'.read'.x, c".write".2x> ∧
   tl(c".write") $\leq^1$ tl(double*(c'.read'))
   ⇒ c".write" $\leq^1$ double*(c'.read') -- Logic, ≤
 1.4 infer c?(c'.read', x) → c!(c".write", 2x) → X
  sat c".write" $\leq^1$ double*(c'.read') -- L3,1.4.1,1.4.2,1.4.3
1: infer P sat c".write" $\leq^1$ double*(c'.read') -- L6, P, 1.3, 1.4
2: from c' = μX.(c'.write'?x → c'.read'!x → X)
 2.1: STOP sat (tr=<>) -- L4
 2.2: tr=<> ⇒ c'.read' $\leq^1$ c'.write' -- Logic, $\leq^n$
 2.3: STOP sat c'.read' $\leq^1$ c'.write' -- L3, 2.1, 2.2
 2.4: from X sat c'.read' $\leq^1$ c'.write'
  2.4.1: c'.write'?x → c'.read'!x → X
   sat (tr ≤ <c'.write'.x, c'.read'.x> ∨
   (tr ≥ <c'.write'.x, c'.read'.2x> ∧
   tl(c'.read') $\leq^1$ tl(c'.write'))) -- L4, 2.4
 2.4.2: tr ≤ <c'.write'.x, c'.read'.x> ⇒ c'.read' $\leq^1$ c'.write' -- Logic, ≤
 2.4.3: tr ≥ <c'.write'.x, c'.read'.2x> ∧
  tl(c'.read') $\leq^1$ tl(c'.write') ⇒ c'.read' $\leq^1$ c'.write' -- Logic, ≤
 2.4 infer c'.write'?x → c'.read'!x → X sat c'.read' $\leq^1$ c'.write' -- L3, 2.4.1,2.4.2,2.4.3
2: infer c' sat c'.read' $\leq^1$ c'.write' -- L6, c', 2.3, 2.4
3: c" sat c".read" $\leq^1$ c".write" -- Similarly to 2
4: (P || c") \ { c | c ∈ αc".write" }
 sat ∃ s • s $\leq^1$ double*(c'.read') ∧ c".read" $\leq^1$ s -- L8, 1, 3
5: (c' || (P || c") \ { c | c ∈ αc".write" }) \ { c | c ∈ αc'.read' }
 sat ∃ t • t $\leq^1$ c'.write' ∧ ∃ s • s $\leq^1$ double*(t) ∧ c".read" $\leq^1$ s
 -- L8, 4, 2
6: ∃ t • t $\leq^1$ c'.write' ∧ ∃ s • s $\leq^1$ double*(t) ∧ c".read" $\leq^1$ s ⇒
  c".read" $\leq^3$ double*(c'.write') -- Logic, $\leq^n$, arith.
7: (c' || (P || c") \ { c | c ∈ αc".write" ∨ c ∈ αc'.read' }
 sat c".read" $\leq^3$ double*(c'.write') -- L3, 5, 6, simplify
infer (c' || P || c") \ {c | c = *write".* ∨ c = *read'.* }
 sat c".read" $\leq^3$ double*(c'.write') -- 7, notation change

Figure 5.2: Proof of the Correctness of Case Study

5.3. Conclusions

MASCOT has been given a denotational semantics structured around its abstract syntax using CSP. Edge labelled, neighbourhood controlled embedding graph grammars have proved to be a very suitable formalism for defining the abstract syntax of graphical MASCOT designs, thus enabling the semantic model to share the same structure as the design. It is claimed that semantic models which share this structure are likely to be more intuitive that those structured around the abstract syntax of a textual representation of MASCOT designs.

A significant core of MASCOT now has a formal underpinning and a supporting theory associated with it: a theory which is standard and well understood. It has been demonstrated how formal MASCOT designs can be proven to be safe with respect to predicate-over-traces specifications, and it has been discussed how they can be proven to be free of deadlock.

In the first part of the thesis, MASCOT, an important informal graphical design method used in industry, has been given a formal semantics. However, the formal specification notation proposed, predicates-over-traces, is quite dissimilar to any informal notations currently used in industry to specify MASCOT designs, and predicates-over-traces specifications do not have a suitable graphical presentation. In the next chapter, an attempt is made to formalise a "mode-based" graphical notation which is sometimes used in industry to specify MASCOT designs.

Chapter 6: Specification Transition Systems

This chapter presents a formal notation for the specification of reactive systems[29]. The notation, known as Specification Transition Systems, (STSs), is mode-based[30]. The semantics of STSs are defined in terms of sets of valid histories or behaviours. A refinement calculus for STSs is defined, and the relationship between STSs and process-based designs is discussed. A way of presenting STSs graphically is described. The chapter is motivated by a discussion of the informal use of modes to specify process-based designs, such as may be described using formal MASCOT.

6.1. Modes and Process-Based Designs

MASCOT designs provide a data-flow perspective of a system, and gives rise to process-based implementations. An alternative perspective of a system can be gained by viewing it as operating in a number of different modes, the system switching between those modes under certain conditions, such as the occurrence of specific events. Modes have been found to be a powerful concept for organising the description of a system.

It is often the case that the modal perspective of a system is a useful abstract characterisation of it. Modal descriptions are therefore sometimes used as a specification of a system, even of systems which are ultimately designed and implemented in a process-based paradigm, such as MASCOT. This, for example, is sometimes the case with BAe Dynamics products, where they have been found to simplify significantly the informal description of systems.

As an example of this way of working, consider a guided missile system. It may be viewed as consisting of four basic components: a launcher, a radar, a missile, and a user-interface. Within the launcher it can be considered that there is a controller, the embedded computer-based system which implements a process-based software design for controlling the missile system. This is depicted in Figure 6.1.

Such a system may be viewed as operating in six basic modes: searching; firing; collecting; guiding; stalled; and warning. In the searching mode, the controller drives the

[29] A discussion of reactive systems can be found in Appendix Two.

[30] What is meant by a "mode-based" notation is clarified subsequently.

radar so that it scans the sky for targets. The event of the radar detecting a target moves

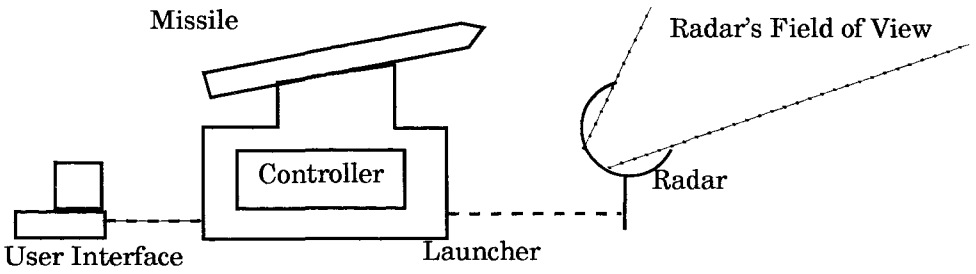


Figure 6.1: Schematic of a Surface-To-Air Missile System

the system into the firing mode. In this mode the controller asks the launcher to fire the missile. Confirmation of the firing from the launcher sends the system into the collecting mode, and failure to receive any confirmation causes the system to enter the stalled mode. In the stalled mode the system alerts the user of the problem, and does nothing until told to continue by the user. In the collecting mode, the controller drives the radar in such a way that it looks for the missile that has been fired. The detection of the missile moves the system into the guiding mode, where the controller sends guidance commands to the missile via the radar to direct it to the detected target. Failure to detect the missile moves the system to the warning mode, where the user is informed of the problem. The interception of the target, or the loss of the missile returns the system to the searching mode, and new targets are sought. The mode and transition information in this description is depicted graphically in Figure 6.2. Obviously, a real system would also contain other modes, such as "off", "test", and "set-up" modes, and other transitions, such as "target-lost". However, those given are sufficient to demonstrate the basic idea of this approach.

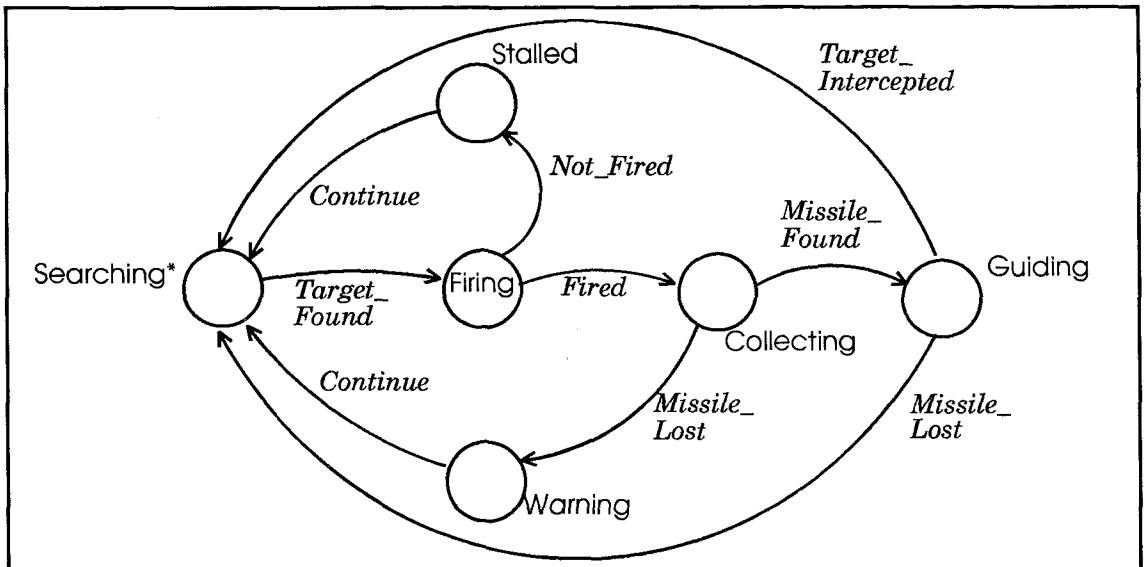


Figure 6.2: A Mode Description of the Missile System

A MASCOT design[31] of the missile system, is given in Figure 6.3. It is loosely based upon a BAe Dynamics' product which was specified informally using a mode-machine.

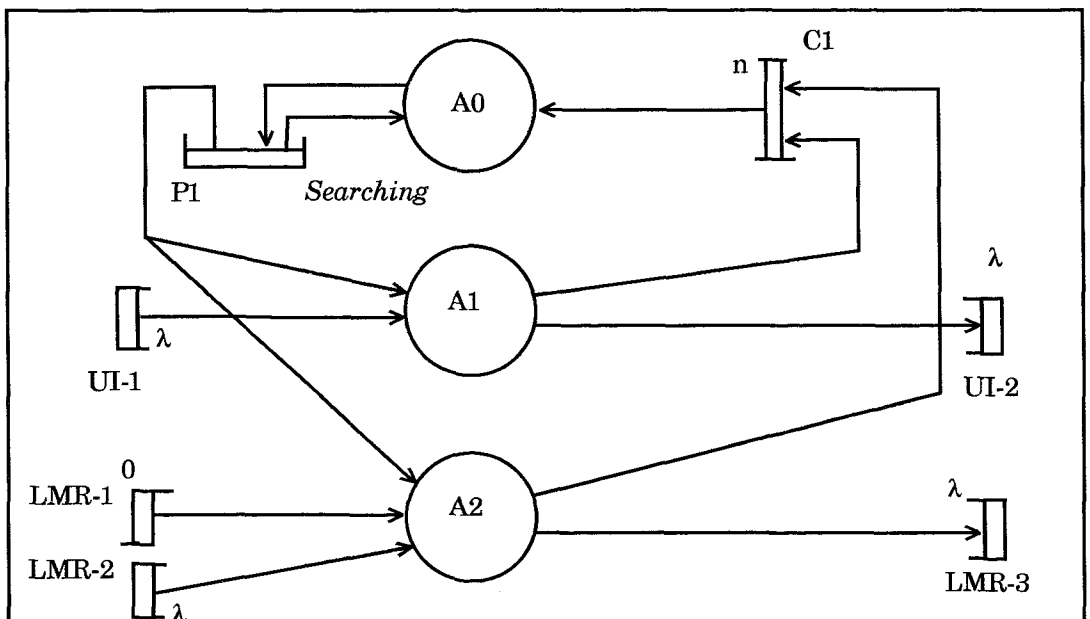


Figure 6.3: A MASCOT Design of the Missile System

The pools, UI-1 and UI-2, model the input and output interface between the controller and the user interface. The pools LMR-2 and LMR-3 represent the input and output passive

[31]The design is not a BLM_GG design, in that it contains a multiple reader pool and a multiple writer channel, but it is a well-formed MASCOT design according to the criteria given in Chapter 2. A basically equivalent BLM_GG design could be given, but it would need more components.

interfaces with the launcher, missile and radar. LMR-2 provides access to the sensors, and LMR-3 provides access to the actuators. LMR-1 is the interface through which active signals from the missile system, such as "Target_Found", reach the controller.

Pool P1 stores a representation of the current mode of the system. This mode information is used by activities A1 and A2 to determine what processing they should be performing. P1 is updated by the activity A0. A0 changes the mode stored in P1 upon receipt of events passed to it through the channel C1, in accordance with the information recorded in the mode-machine.

The basic algorithm structure of activity A1 is given in pseudo-code in Figure 6.4. The algorithm structure for A2 is similar to A1, but instead of executing periodically, delaying, and polling the pool P1, it consumes events from the signal LMR-1, executing immediately a signal arrives. A2 is an aperiodic or sporadic activity. The algorithm structure for A2 is given in Figure 6.5.

```
Activity A1:
    ....                                -- any initialisation code needed
    loop
      p?( P1, current_mode );           -- A1 takes copy of the mode from the pool P1
                                        -- A1 is now committed to processing in this mode
      p?( UI-1, user_input );           -- Commands read from user interface
      c!( C1, user_input );             -- Events are passed to A0, the mode controller
      case current_mode of
        Searching   : ....              -- Outputs are produced for the user
        Firing      : ....              -- appropriate to the view A1 has of the mode
        Guiding     : ....              -- of the system
        ....
      end case;
      delay X;                          -- A1 is a periodic process. With some scheduling
                                        -- implementations a "suspend" might be
                                        --  preferable instead.
    end loop                            -- A1 services the user interface continuously
End Activity;
```

Figure 6.4: Algorithm for Activity A1

```
Activity A2:
    ....                                -- any initialisation code needed
    loop
      s?( LMR-1, event );               -- wait for, then store, any signals from LMR-1
      p?( P1, current_mode );           -- A2 takes copy of the mode from the pool P1
                                        -- A2 is now committed to processing in this mode
      p?( LMR-2, status_of_LMR );       -- sensors read from LMR-2
      c!( C1, events );                 -- Events are calculated from LMR event and
                                        -- status, and are passed to A0 via C1.
      case current_mode of
        Searching   : ....              -- Outputs are produced for the launcher, missile,
        Firing      : ....              -- and radar appropriate to the view A2 has of the
        Guiding     : ....              -- mode of the system.
        ....
      end case;
    end loop;                           -- A2 loops, ready to process more events from the
                                        -- missile system, should any have arrived.
End Activity;
```

Figure 6.5: Algorithm for Activity A2

This missile example is discussed further later on in the chapter. For now, it simply serves as an example of a mode-based specification of a process-based design. A number of points should be noted about this informal use of mode.

Firstly it should be noted that, although these mode-machines seem similar to Moore Finite State Machines[32] (FSMs), they are different. A Moore FSM maps a state to an output, while a mode-machine associates an output process with a mode, or, alternatively, a sequence of outputs with a mode. This difference is not always noted, and this is probably why, for example, Avnur says that Moore FSMs have proven themselves as an intuitive and useful specification approach in software engineering, [Avn90].

Secondly, it should be noted that modes are not mutually exclusive like states. States can be viewed as defining the exhaustive set of "things" in which a system resides, and of requiring that at any time a system is in one, and only one, state. Modes by contrast, it can be argued from the above example, are not so restrictive. The activities A1 and A2 may concurrently perform processing appropriate to different modes due to delays in becoming aware of a mode changes in P1. This toleration of some "raggedness" throughout the system in transition from mode to mode means that modes should be carefully distinguished from states.

[32] A Moore FSM is a finite state machine with output, where the outputs are associated with a state; Mealy FSMs associate outputs with transitions, [HoU79].

These two differences between mode-machines and state-machines are important because it is these differences which make it practical to describe large systems using modes. Mode-machines essentially become shorthand notations for much larger state-machines: models which have a much more manageable state-space, and which more closely correspond to an intuitive view of the system. It is, for example, possible to map CSP described process-based designs into state-machines. Each process can be viewed as a state-machine, and the state-space for the state-machine which corresponds to the parallel composition of such processes is essentially the cross-product of the state-spaces of each process, [Jos88]. Unfortunately for designs of any size, this results in an explosion in the size of the state-space. Such state-machines are not the abstract mode-based machines which it is helpful to construct in specifying a system.

To summarise; as far as this chapter is concerned, mode-machines differ from state-machines in two ways: 1) a mode-machine may produce a series of outputs while residing in a mode, while a state-machine defines an output per state; and 2) at any given instant, a mode-machine may be in more than one mode at a time, while a state-machine must be in only one state.

In this chapter an attempt is made to formalise mode-machines. The model defined is named a Specification Transition System or an STS.

STSs are founded on the observation that the essence of the mode abstraction over state lies in the freedom it gives to the designer to build a design which consists of elements which do not change mode synchronously. This freedom is usually only implicit in an informal mode-machine specification, but nevertheless is normally understood to be limited. A correct design will not contain a subsystem which stays in the old mode for "too long". Sometimes this limitation is made explicit in terms of temporal bounds on the maximum delay between producing certain outputs after entering a given mode. Such temporal bounds normally reflect the fact that the outputs are being sent to devices which the system is controlling and which need to be driven in a mutually consistent way. That is, the devices should share the same perception of the system's current mode. In the missile system, for example, it could be contended that the missile and the launcher are two devices which need a consistent view of the system's mode, especially when the mode is Firing.

No attempt will be made to provide quantitative temporal notations or semantics for STSs. Instead, the devices which an STS system controls will be identified, and any which need to be driven consistently will be grouped together. A constraint will be imposed which

requires that, in a given mode, if one device has an output sent to it, all the others devices in the same group must also receive outputs appropriate for the same mode. It is believed that, by only imposing this constraint on devices in the same group, the freedom which makes modes such a useful abstraction is obtained. This point will be returned to after STSs have been defined.

6.2. Specification Transition Systems

In this section STSs are presented. They are based around Labelled Transition Systems (LTSs), [Hen88], [Mil89], and [WiN92], but have extensions and constraints to make the "states" more like the modes of a mode-machine. However, it will be argued that these extensions do not extend the fundamental expressive power of LTSs.

An STS is a seven tuple: $<M, M_0, A, T, S, D, \delta>$, where

M = set of system modes.

M_0 = set of initial system modes, where $M_0 \subseteq M$ and $M_0 \neq \emptyset$.

A = set of actions in which the system can engage. Actions are classified as being of one of three kinds: inputs, events, and outputs, denoted by the sets Ip, Ev, and Op respectively. Formally, $A = Ip \cup Op \cup Ev$, and Ip, Op and Ev are disjoint.

T = a relation that defines the transitions between modes in which the system can engage. $T \subseteq M \times A \times M$[33], where the first M defines the start mode; the A defines the actions that occurs upon that transition; and the second M defines the end mode. It is assumed that there are functions Actions, Start and End, which return the appropriate element of a transition.

Outputs may only label transitions which start and end on the same mode:

$$\forall m, m' \in M, a \in A \bullet (a \in Op \wedge m \rightarrow^{a} m') \Rightarrow m = m'.$$

S = the set of devices outside the system.

D = the destination function, which maps transitions labelled with outputs to sinks. $D : T \rightarrow S$. All output labelled transitions must be mapped to a sink. Also only one transition per mode may be mapped to a sink, and a transition may only be mapped to one sink. Formally:

$$\forall t \in T \bullet t \in \text{dom } D \Leftrightarrow \text{Action}(t) \in Op$$

$$\forall t, t' \in T \bullet D(t) = D(t') \wedge \text{Start}(t) = \text{Start}(t') \Rightarrow t=t'$$

---

[33] For convenience, the abbreviation "TRANSITION" will be used for $M \times A \times M$.

[34] $m \rightarrow^{a} m'$ will be used as short-hand for $(m, a, m') \in T$.

δ = the dependency relation between sinks. $\delta \subseteq S \times S$. Two sinks are defined by the specifier as being dependent by relating them via δ.

The sets M, M_0, A and T are as for LTSs, except that A is not normally partitioned into three subsets, and transition labels are unconstrained. The partitioning of actions is done to invoke the intuitions of the specifier, and to enable the constraint on outputs to be stated, which requires that they only label transitions which start and end in the same mode. This constraint is imposed on output labels to reflect the fact that output processes are associated with modes and not transitions in informal mode-machines.

S, D and δ are totally new elements of the tuple. S is the set of sinks, and should be considered to define the devices outside the system, to which output values are sent. D maps the output labelled transitions to the sinks, and δ defines that certain sinks are "dependent". The intended intuition is that sinks which are dependent upon each other are a critical resource which needs to be updated consistently in each mode. Formally, δ imposes a constraint on the ordering of transitions which can occur in valid histories in which the STS can engage. This is formalised below.

There are also certain properties or constraints that must apply to the static definition of the dependency relation, given its intended intuition. Clearly, every sink is "dependent" upon itself; also, if one sink is dependent upon a second, the second must also be dependent upon the first; and lastly, if one sink is dependent upon a second, which in turn is dependent upon a third, the first is also dependent on the third. Formally this means that δ is a reflective, symmetric, and transitive relation. That is:

$$\forall s, s', s'' \in S \bullet$$
$$s \,\delta\, s \wedge$$
$$s \,\delta\, s' \Rightarrow s' \,\delta\, s \wedge$$
$$s \,\delta\, s' \wedge s' \,\delta\, s'' \Rightarrow s \,\delta\, s''.$$

The fact that these properties must be true of δ, means that δ forms a "dependency" equivalence class on Sinks. The following notation will be used to denote the subset of sinks in the same dependency class as a given sink, s: "∂_s", where $\partial_s = \{ s' \in S \mid s \,\delta\, s' \}$.

Special event actions, known as *output commitments*, are also identified. The set of all such events is denoted by OC, where $OC \subseteq Ev$. These events have names, a field of which is the name of a sink. A function, Committed_Sink: $OC \rightarrow S$, is defined for such events, it returns the sink to which "commitment" has been made. Output commitment event actions,

like output actions, may only label transitions whose start and end modes are identical. Formally:

$$\forall m, m' \in M, a \in A \bullet (a \in OC \wedge m \rightarrow^{a} m') \Rightarrow m = m'$$

The definition of the valid histories that an STS can engage in will reveal that output commitments are actions which commit the STS to produce outputs directed to all the sinks in the same class as the sink to which commitment has been made; outputs which are appropriate for the mode in which the STS was residing when the commitment was made. Outputs will only occur after a suitable commitment has been made. Strictly, per mode, it is only necessary for there to be one transition, per class of sinks, labelled with an output commitment, but this is not stated as a constraint on the construction of STSs.

6.2.1. Some Definitions

An STS in which M_0 is a singleton set, and for every mode, there is only one transition in T which starts from that mode and is labelled with that action, is known as a *deterministic* STS. A deterministic STS is therefore one which satisfies the following property:

$$card(M_0) = 1 \wedge ((m \rightarrow^{a} m' \wedge m \rightarrow^{a} m'') \Rightarrow m' = m'')$$

An STS which does not satisfy this property is known as a *non-deterministic* STS.

It should be noted that, like LTSs, the behaviour of a deterministic STS is determined by the sequence of actions that it engages in. Thus deterministic STSs are *unambiguous*, [Kwi89].

All modes should be *reachable* from an initial mode. This is the weakest constraint that ensures that all the modes contribute to the description of the system. A mode is reachable if there is a sequence of transitions from an initial mode to that mode. Formally this constraint can be defined as:

$$\forall m \in M \bullet \exists m' \in M_0 \bullet Path(m', m)$$

where Path is defined as:

Path(m1, m2 : M) b : Boolean
post
b $\Leftrightarrow$ m1 = m2 $\vee$

$$\exists\, m \in M \bullet$$
$$\exists\, a \in A \bullet (m1, a, m) \in T \wedge Path(m, m2)$$

It should be noted that there has been no requirement for there to be transitions labelled with every input, in every mode. An STS which satisfies such a requirement is called an *Input Complete STS*. Formally, an Input Complete STS satisfies:

$$\forall\, m \in M, i \in Ip \bullet \exists\, t \in T \bullet Start(t) = m \wedge Action(t) = i$$

If an STS, in every mode, defines an output to every sink in the dependency classes to which it can make an output commitment, it is called *Output Complete*. Formally:

$$\forall\, m \in M \bullet$$
$$\exists\, t \in T \bullet$$
$$Start(t) = m \wedge Action(t) \in OC \Rightarrow$$
$$\forall\, s \in S \bullet$$
$$s \in \partial_{Committed\_Sink(Action(t))} \Rightarrow \exists\, t' \in T \bullet Start(t') = m \wedge D(t') = s$$

If an STS further defines, for each mode, an output commitment to each dependency class, the STS is called *Fully Output Complete*. Formally:

$$\forall\, m \in M, s \in S \bullet \exists\, e \in OC \bullet Committed\_Sink(e) \in \partial_s \wedge m \rightarrow^e m$$

An STS which is Input and Output Complete will be called a *Complete* STS. It is suggested that it is normally best to ensure that an STS which is used for specification is Complete.

6.2.2. Dynamic Behaviour

A *serial derivation* of an STS, $\Sigma \equiv \langle M, M_0, A, T, S, D, \delta \rangle$, is a sequence of transitions which only contain transitions drawn from T, and where the start of the first transition in the sequence is a member of M_0, and where the end and start of adjacent transitions in the derivation coincide. Der*(Σ) is the set of all finite serial derivations for the STS, Σ. Formally:

$$Der^*(\Sigma) \equiv \{\, d : TRANSITION\text{-}seq \mid$$
$$Start(d_0) \in M_0 \wedge \forall\, i \in 0 \,..\, Len(d) \bullet d_i \in T \wedge Is\_Serial(d) \,\}$$

where, for a sequence s, s_i represents the i^{th} element of the sequence, and

Is_Serial is a function, with signature Is_Serial: TRANSITION-seq → BOOLEAN, whose definition is:

Is_Serial(d : TRANSITION-seq) b : BOOLEAN

post b ⇔ ∀ i ∈ 0 .. Len(d) - 1 • End(d_i) = Start(d_{i+1})

Before the behaviour of an STS can be defined, input-event serial derivations (or "IES derivations") must also be defined. An IES derivation for an STS, Σ, is a sequence of transitions all of which have been defined in the T element of Σ. Unlike serial derivations the end and start modes of adjacent transitions in the derivation need not coincide. In fact, only transitions labelled with either input or event actions need be serial, while output labelled transitions do not need to be "in sequence". The set of all IES derivations for an STS, Σ, is denoted by IES_Derivations(Σ). Formally:

IES_Derivations(Σ) ≡ { d : TRANSITION-seq |

Start(d_0) ∈ M_0 ∧ ∀ i ∈ 0 .. Len(d) • d_i ∈ T ∧ Is_Serial(d ⌈ (Ip ∪ Ev)) },

where ⌈ is the sequence restriction operator. Informally it removes from a sequence of transitions all transitions not labelled with actions from the second parameter. Formally:

the signature of ⌈ is: infix ⌈ : TRANSITION-seq × A → TRANSITION-seq

and the semantics of ⌈ is defined by:

<> ⌈ S = <>

t ⌈ S = head(t) ^ (tail(t) ⌈ S) if action(head(t)) ∈ S.

t ⌈ S = tail(t) ⌈ S if action(head(t)) ∉ S.

where ^ is the sequence concatenation operator, and <> is the null sequence.

The set of behaviours or histories in which an STS may engage is larger than the set of serial derivations, but is smaller than the set of IES derivations. If the set of behaviours of an STS, Σ, is denoted by Beh(Σ), this can be formalised as:

Der*(Σ) ⊂ Beh(Σ) ⊂ IES_Derivations(Σ).

Informally, the rationale for Der*(Σ) ⊂ Beh(Σ) is that once an STS makes a commitment to a sink in a given mode, the STS must engage in all the transitions

labelled with outputs defined on that mode which are mapped by D to a sink in the same dependency class as the sink to which the commitment has been made. However, these outputs may occur after the STS has moved on to a new mode. Thus not all behaviours are serial, and in this way the aspect of modes not being mutually exclusive like states is modelled. However, $Beh(\Sigma) \subset IES\_Derivations(\Sigma)$ holds, because outputs are not totally unconstrained in the order in which they may occur. Only outputs directed to sinks in a class to which a commitment has been made may occur "out of sequence", and then only once, without a subsequent output commitment. Output Commitments, being event actions, must occur serially. The constraint will be imposed that, subsequent output commitments to a sink in a dependency class may not occur until after all the previous outputs to that class have occurred.

The precise restrictions which derivations in $Beh(\Sigma)$ must satisfy are now formalised.

$$Beh(\Sigma) = \{ d : TRANSITION\text{-}seq \mid STS\_Der(d,\Sigma) \}$$

where

$STS\_Der(d,\Sigma) \Leftrightarrow$

$d \in IES\_Derivations(\Sigma) \wedge$

$\forall i \in N \bullet$

$i \leq Len\ d \wedge$

$Action(d_i) \in Op \Rightarrow$

-- the destination of the output is to a sink which has not been written to

-- since its dependency class was last enabled in d. And the output is as

-- defined for the mode in which the commitment was made.

$\exists j \in N \bullet$

$j < i \wedge Action(d_j) \in OC \wedge Committed\_Sink(Action(d_j)) \in \partial_{D(dj)} \wedge$

$\neg \exists k \in N \bullet j<k<i \wedge D(d_k) = D(d_i) \wedge$

$Start(d_i) = Start(d_j) \wedge$

$Action(d_i) \in OC \Rightarrow$

-- the commitment is to a dependency class not currently enabled in d; (d

-- $\in$ IES_Derivations ensures that the commitment is appropriate for the

-- given mode.)

$\exists j \in N \bullet$

$j < i \wedge Action(d_j) \in OC \wedge$

$Committed\_Sink(Action(d_j)) \in \partial_{Committed\_Sink(Action(di))} \wedge$

$$
\begin{aligned}
&\neg \exists k \in N \bullet \\
&\quad j<k<i \wedge Action(d_k) \in OC \wedge \\
&\quad Committed\_Sink(Action(d_k)) \in \partial_{Committed\_Sink(Action(di))} \wedge \\
&\forall sink \in \partial_{Committed\_Sink(Action(di))} \bullet \\
&\quad \exists l \in N \bullet \\
&\qquad j<l<i \wedge D(d_l) = sink \\
\vee \\
&\neg \exists j \in N \bullet \\
&\quad j < i \wedge Action(d_j) \in OC \wedge \\
&\quad Committed\_Sink(Action(d_j)) \in \partial_{Committed\_Sink(Action(di))}
\end{aligned}
$$

It should be noted that Beh(Σ) is the set of possible transitions that an STS can engage, however, the transitions are not observable, only the sequences of actions associated with the transitions. Obs_Beh(Σ) is the set of observable behaviours, and is defined as:

$$Obs\_Beh(\Sigma) = \{ b : A\text{-}seq \mid \exists d \in Beh(\Sigma) \bullet \forall i \in 0 .. len(d) \bullet Action(d_i) = b_i \}.$$

6.2.3. Intuitions about STSs

It is intended that transitions labelled with input actions are transitions that occur when an input occurs in the system's environment. It is also intended that transitions labelled with outputs are those which, when the system engages in them, send an output value to the system's environment. It should be noted that there is no synchronisation implied between the environment and the system. It is intended that the transitions labelled with actions which are not inputs can occur at any time the STS is in the mode from which the transition starts; that is, the the environment may not block the STS.

An input which occurs while the STS is in a mode which is not the start mode of a transition labelled with that input, is considered to be ignored by the STS. It would be unrealistic to expect a system's environment to freeze with the input continually offered until the STS is ready to process it. Likewise, the environment cannot prevent the system from engaging in a transition labelled with an output. This follows Murphy's classical observations and concurrency theory, [Mur91a], [Mur91b] and [MuP91], where observations of a system are non-interactive, unlike for example, many process algebras,

which assume that to observe what a system is doing there must be some interaction with it, [Hoa85], [Hen88], and [Mil89].

Transitions labelled with event actions correspond to autonomous, internal, actions of the STS, which involve no exchange of data with the environment. Some event names will be evocative of situations, such as a particular time-out. However STSs do not provide any way of recording or formalising these intuitions. Hence it will be the task of the validator to convince himself (informally) that the implementation of an event will only occur when the conditions implicit in the event's name arise. Other event names will be evocative of a computation or process, similar to output actions' names.

Output names will normally be evocative of an output process rather than an explicit output value. The intuition is that such an event or output action will be refined into a process in the design. Unfortunately, however, there is not an adequate definition of action refinement in semantic domains such as have been used for STSs, [Ace92]. This will be discussed further in Section 6.6.3.

Before STSs are considered further, a graphical presentation scheme is introduced for them.

6.3. Presenting STS Specifications

So far, STSs have been presented as basic mathematical structures with an associated formal semantics. However, it is not usually convenient to work directly with mathematical structures; a suitable presentation scheme can significantly contribute to making the same information more accessible. Hence a graphical presentation for STSs is described in this section. It is normally the case that people can develop better intuitions about graphically presented systems, and this has motivated other visual formalisms, for example [Har87] and [Har88]. A suitable presentation approach can also sometimes help to encourage the construction of well-formed structures.

It has long been standard when graphically presenting state-machines and automata to use circles or dots to represent states; arrows between states to represent transitions, and text labels to define the actions. The same basic presentation scheme will be used for STSs, however of course, it will be extended to handle sinks, sink dependencies, and the mapping of output labelled transitions to sinks. Sinks will be presented by square boxes, which will be positioned to one side of the diagram in a single vertical line. They will be ordered so that

all the sinks in the same dependency class will be contiguous. The classes will be represented by a dashed line between adjacent sinks in the same class. Thus, the reflective, symmetric, and transitive aspects of the dependency relation will not be presented graphically, but should be considered to be implicitly present. This significantly reduces the dependency lines which need to be drawn.

The names of all non-output actions should be written adjacent to the arrow which represents the transition with which they are associated. To reduce the number of arrows, two or more actions may label the same arrow, being separated using the "+" symbol. Input and event actions should be distinguished on the diagram by the use of a postfix subscript "i" or "e".

The name of an STS mode is also presented by text. This is placed either within, or adjacent to, the circle representing that mode. Initial modes of an STS are identified by a star, "*", after the mode name.

Rather than follow the "obvious" approach of treating output labelled transitions like the rest, and also adding lines linking each output to its sink, use is made of the constraints imposed on the positioning of sinks to keep the presentation simple. Output actions are ordered in vertical vectors adjacent to the mode with which they are associated. The position of an output in a vector corresponds to the sink to which it is directed. That is, the i^{th} output in a vector is mapped to the i^{th} sink in the column of sinks at the side of the diagram.

The definition of an STS is such that it is not possible to distinguish two transitions labelled with the same output defined on the same mode, even if the output is being directed to different sinks. It may sometimes be desirable to send the same output to different sinks, in the same mode, so the convention will be adopted that the same output value may occur more than once in the same output vector, it being implicit that there is more to the output names which distinguish them.

Redundant, but helpful, information is provided by the use of dashed lines to separate the values in the output vectors into the dependency classes. Thus, all the elements of the vector not partitioned by a dashed line are directed to sinks in the same dependency class, and so will be updated "together". A dash is placed in an output vector where no output is defined for that sink, in that mode.

An example of such a presentation of an STS is:

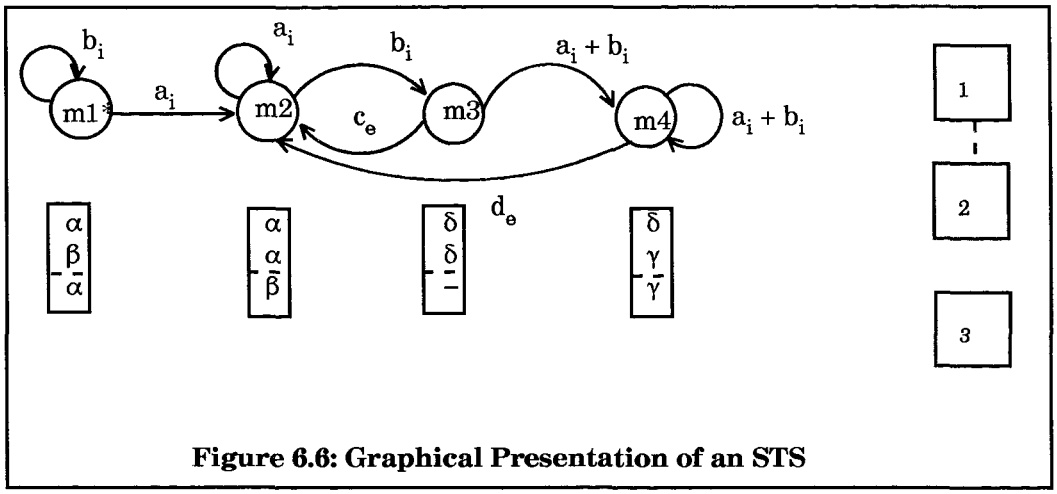


Figure 6.6: Graphical Presentation of an STS

The STS portrayed above is: <M, M_0, A, T, S, D, δ>, where

M = { m1, m2, m3, m4 }

M_0 = { m1 }

A = { a, b, c, d, oc1, oc3 α1, α2, α3, β, γ2, γ3, δ1, δ2 } ,

where Ip = { a, b }, Ev = { c, d, oc1, oc3 }, and Op = {α1, α2, α3, β, γ2, γ3, δ1, δ2}

T = { m1 $\rightarrow^{a}$ m2, m1 $\rightarrow^{b}$ m1, m1 $\rightarrow^{\alpha 1}$ m1, m1 $\rightarrow^{\beta}$ m1, m1 $\rightarrow^{\alpha 3}$ m1,
m2 $\rightarrow^{a}$ m2, m2 $\rightarrow^{b}$ m3, m2 $\rightarrow^{\alpha 1}$ m2, m2 $\rightarrow^{\alpha 2}$ m2, m2 $\rightarrow^{\beta}$ m2,
m3 $\rightarrow^{a}$ m4, m3 $\rightarrow^{b}$ m4, m3 $\rightarrow^{c}$ m2, m3 $\rightarrow^{\delta 1}$ m3, m3 $\rightarrow^{\delta 2}$ m3,
m4 $\rightarrow^{d}$ m2, m4 $\rightarrow^{b}$ m4, m4 $\rightarrow^{a}$ m4, m4 $\rightarrow^{\delta 1}$ m4, m4 $\rightarrow^{\gamma 2}$ m4,
m4 $\rightarrow^{\gamma 2}$ m4, m1 $\rightarrow^{oc1}$ m1, m1 $\rightarrow^{oc3}$ m1, m2 $\rightarrow^{oc1}$ m2, m2 $\rightarrow^{oc3}$ m2,
m3 $\rightarrow^{oc1}$ m3, m4 $\rightarrow^{oc1}$ m4, m4 $\rightarrow^{oc3}$ m4 }

S = { 1, 2, 3 }

D = { m1 $\rightarrow^{\alpha 1}$ m1 ⇒ 1, m1 $\rightarrow^{\beta}$ m1 ⇒ 2, m1 $\rightarrow^{\alpha 3}$ m1 ⇒ 3, m2 $\rightarrow^{\alpha 1}$ m2 ⇒ 1,
m2 $\rightarrow^{\alpha 2}$ m2 ⇒ 2, m2 $\rightarrow^{\beta}$ m2 ⇒ 3, m3 $\rightarrow^{\delta 1}$ m3 ⇒ 1,
m3 $\rightarrow^{\delta 2}$ m3 ⇒ 2, m4 $\rightarrow^{\delta 1}$ m4 ⇒ 1, m4 $\rightarrow^{\gamma 2}$ m4 ⇒ 2,
m4 $\rightarrow^{\gamma 3}$ m4 ⇒ 3 }

δ = { 1 δ 2 }

Note that this STS is not Fully Output Complete because there is no output defined for sink 3, in mode m3. This is illustrated on the diagram by a dash in the appropriate point in the output vector for mode m3. To determine if the STS is Output Complete or not, depends on whether or not there is a transition on the mode which is labelled with an output commitment to a sink in the class of which there is not an output. The static requirements

are such that if there are outputs to other sinks in the same dependency class as the one missing this will be obvious, and the STS will not be Output Complete. Adopting the convention that, on any given mode, there are only output commitments to sinks in dependency classes for which there are other outputs to sinks that class, on that mode, has the advantage that Output Completeness, as well as Fully Output Completeness, can be determined from the diagram. In the STS above, adopting this convention, it is Output Complete, but not Fully Output Complete. Thus output commitments are not shown explicitly. Their explicit presentation would not reveal any new information about the system.

It can also be helpful to present the mode and transition information in a transition table. The modes of the STS label the columns, the actions label the rows, and the contents of the table are, usually singleton, sets of mode names; the names represent the termination mode of a transition labelled with the action of the row, which starts from the mode of the column. The transition table for the above example is given below. Note that the brackets around singleton sets have been dropped for convenience.

		Current Mode			
	Next Mode	m1*	m2	m3	m4
Transition	a_i	m2	m2	m4	m4
	b_i	m1	m3	m4	m4
	c_e	-	-	m2	-
	d_e	-	-	-	m2

Figure 6.7: Transition Table

The dashed line separates input labelled transitions from event labelled transitions, and the dashes within the table indicate that no transition is defined with that action from that mode. An STS which has no dashes associated with input labelled transitions, as in the above example, is Input Complete.

There is no advantage in including transitions labelled with outputs or output commitments in such a table; they always label transitions which return to the same mode. Also, there is no advantage in recording the output information in a different table, for example one which records the output values for each sink and mode. Such a table would simply be repeating the output vectors drawn on the STS diagram, in essentially the same form.

6.4. STSs and LTSs

Labelled Transition Systems (LTSs) are defined by the tuple: <S, S_0, A, T>, where S is a set of states; S_0 is the set of initial states, $S_0 \subseteq S$; A is a set of actions or events; and T is transition relation: T : S × A × S, which relates states and actions to a successor state. (Sometimes slightly different definitions are given: for example, [Hen88], [Mil89], and [Nie92], but they are all essentially the same.)

It has been contended that STSs have not changed the expressive power of LTS, but have only introduced a more terse presentation scheme, where modes effectively represent an equivalence class of LTS states. It is not contended that every LTS can be represented as an STS, only that any STS can be represented as an LTS. To justify this contention, a mapping is sketched from STSs to LTSs, and it is illustrated with some small examples.

The state-space of an LTS will be significantly larger than the mode-space of the corresponding STS. The names of the states of the LTS may be constructed from the modes of the STS, together with some indication of which classes of sinks have been committed to in that state, and which sinks have been written to. The information recorded in the D mapping, may be recorded in the names of the output actions. Output actions of the LTS should be considered to be divided into fields representing each sink. An output which labelled a transition in an STS which was mapped to a sink, would be represented as an action with blank fields apart from the field corresponding the sink to which it was directed, and this would contain the name of the original output. The information recorded by the δ relation is encoded in an LTS in terms of which states and transitions are present.

A small example is given below:

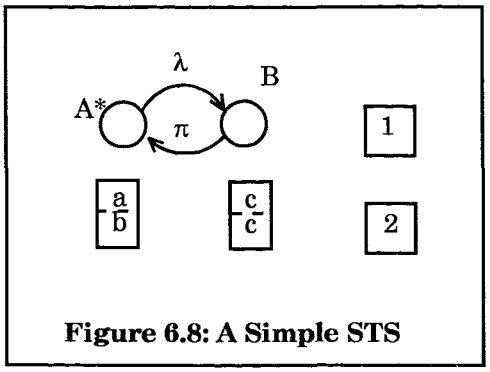


Figure 6.8: A Simple STS

The corresponding LTS has eighteen states. The state names are made up of three fields: one which records the current mode; one which records which mode, if any, has currently made an output commitment to sink 1, and one which records which mode, if any,

has currently made an output commitment to sink 2. Thus the alphabets of each field are: {A, B}, { ∅, A, B } and { ∅, A, B } respectively, where "∅" indicates that there is no outstanding commitment to that sink. The size of the state-space of the LTS is 2 × 3 × 3 or 18. The transition labels of the LTS are the same as for the STS, with the appropriate renaming of outputs. The existence of transitions in the larger state-space of the LTS is determined by the transitions in the STS and the "meaning" of the state names. Thus, for example, the output commitment transitions are only defined on states for which the class of sinks to which they commit has not already been committed to. Thus, oc1 only labels transitions which start from states with a ∅ in the second field of the state name. Such transitions always terminate on states with the same name, except that the second field now has the same name as the current mode, that is, the second field of the state name is identical to the first field of the state name. Similarly, output labelled transitions, only start from states which record the fact that the class of sink to which they are directed has been committed to in the mode in which that output is defined, and that that particular sink has not been written to since then. Thus, in this example, a_ only starts from states with an A as the second field, and _c only starts from starts from states with a B in the third field. In cases where all the other sinks (if there are any) have already been written to, such a transition ends in the state with that field restored to ∅, while in the case where there are other sinks which still need to be written to, the state name changes to record that this sink has now been written to. Input and event labelled transitions change the first field of the state name in the same way that they change the mode of the STS.

The transition table for the LTS which corresponds to the above STS is given in Figure 6.9. The initial state(s) of the LTS is(are) the initial mode(s) of the STS with no output commitments to any sinks, and is(are) distinguished by a * as for STSs.

Current States	Transition Labels							
	oc1	oc2	a_	_b	c_	_c	λ	π
AØØ*	AAØ	AØA	-	-	-	-	BØØ	-
AØA	AAA	-	-	AØØ*	-	-	BØA	-
AØB	AAB	-	-	-	-	AØØ*	BØB	-
AAØ	-	AAA	AØØ*	-	-	-	BAØ	-
AAA	-	-	AØA	AAØ	-	-	BAA	-
AAB	-	-	AØB	-	-	AAØ	BAB	-
ABØ	-	ABA	-	-	AØØ*	-	BBØ	-
ABA	-	-	-	ABØ	AØA	-	BBA	-
ABB	-	-	-	-	AØB	ABØ	BBB	-
BØØ	BBØ	BØB	-	-	-	-	-	AØØ*
BØA	BBA	-	-	BØØ	-	-	-	AØA
BØB	BBB	-	-	-	-	BØØ	-	AØB
BAØ	-	BAB	BØØ	-	-	-	-	AAØ
BAA	-	-	BØA	BAØ	-	-	-	AAA
BAB	-	-	BØB	-	-	BAØ	-	AAB
BBØ	-	BBB	-	-	BØØ	-	-	ABØ
BBA	-	-	-	BBØ	BØA	-	-	ABA
BBB	-	-	-	-	BØB	BBØ	-	ABB

Figure 6.9: Transition Table for the Corresponding LTS

The above example does not illustrate the state naming convention which should be adopted when there are dependency classes of sinks greater than one. Therefore, the same example is repeated with the addition of a third sink which is dependent upon the second. The output vectors are also extended.

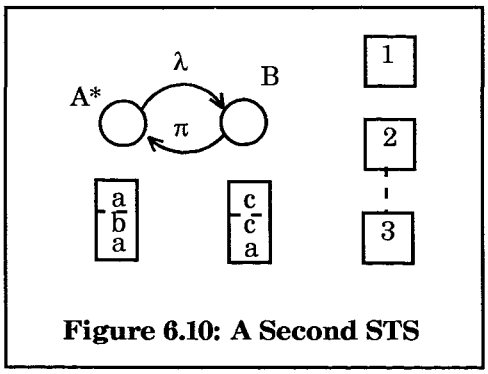


Figure 6.10: A Second STS

There are still three fields to the state name of the corresponding LTS. These fields represent the current mode, and the states of both of the dependency classes of sinks respectively. As before, the alphabets of the first two fields are {A, B} and {Ø, A, B}. However, the alphabet of the third field is extended to enable it to record which sinks in

that class have been written to since the last commitment to it. Its alphabet is {Ø, A, B, A_2, A_3, B_2, B_3}, where, for example, A_2 means that sink 2, but not sink 3, has been written since commitment occurred in mode A, and B_3 means that sink 3, but not sink 2, has been written since commitment occurred in mode B.

This LTS therefore has 2 × 3 × 7 or 42 states. The action alphabet is {oc1, oc23, a__, _b_, __a, c__, _c_, λ, π}. Due to the size of the LTS, only a fragment of its transition table is given in Figure 6.11.

It is informative to consider the implications for the LTS, should an extra dependency be defined between sink 1 and sinks 2 and 3. The state name would be contracted to two fields, the current mode, and the status of the dependency class. The alphabet of the current mode would remain the same, but the alphabet of the second field extends to become: {Ø, A, B, A_1, A_2, A_3, A_{12}, A_{13}, A_{23}, B_1, B_2, B_3, B_{12}, B_{13}, B_{23} }, where, for example, A_{12} represents the state where an output commitment to the class has been made in mode A, and the sinks 1 and 2 (but not 3) have been written to since that commitment. The size of the state-space of the LTS is reduced to 30 by introducing this dependency.

Current States	Transition Labels								
	oc1	oc23	a__	_b_	__a	c__	_c_	λ	π
AØØ*	AAØ	AØA	-	-	-	-	-	BØØ	-
AØA	AAA	-	-	$AØA_2$	$AØA_3$	-	-	BØA	-
$AØA_2$	AAA_2	-	-	-	AØØ*	-	-	$BØA_2$	-
$AØA_3$	AAA_3	-	-	AØØ*	-	-	-	$BØA_3$	-
AØB	AAB	-	-	-	$AØB_3$	-	$AØB_2$	BØB	-
$AØB_2$	AAB_2	-	-	-	AØØ*	-	-	$BØB_2$	-
$AØB_3$	AAB_3	-	-	-	-	-	AØØ*	$BØB_3$	-
AAØ	-	AAA	AØØ*	-	-	-	-	BAØ	-
AAA	-	-	AØA	AAA_2	AAA_3	-	-	BAA	-
AAA_2	-	-	$AØA_2$	-	AAØ	-	-	$BØA_2$	-
AAA_3	-	-	$AØA_3$	AAØ	-	-	-	$BØA_3$	-
AAB	-	-	AØB	-	AAB_3	-	AAB_2	BAB	-
AAB_2	-	-	$AØB_2$	-	AAØ	-	-	BAB_2	-
AAB_3	-	-	$AØB_3$	-	-	-	AAØ	BAB_3	-
ABØ	-	ABA	-	-	-	AØØ*	-	BBØ	-
" "									
BBB	-	-	-	-	BBB_3	BØB	BBB_2	-	ABB
BBB_2	-	-	-	-	BBØ	$BØB_2$	-	-	ABB_2
BBB_3	-	-	-	-	-	$BØB_3$	BBØ	-	ABB_3

Figure 6.11: Transition Table for the Second LTS

These examples have illustrated the construction of an LTS which corresponds to an STS. They have demonstrated that the modes of a system may be significantly fewer than its states. The general relationship between the size of the state-space of an LTS and a corresponding STS is given by the following equation:

$$\text{No\_of\_States} = M \times \alpha S_1 \times \alpha S_2 \times .... \alpha S_n$$

where M is the number of modes, n is the number of dependency classes, and αS_i is the size of the alphabet of the i^{th} dependency class of sinks. The size of this alphabet is given by the following equation:

$$\alpha S_i = \quad 1 + M + \left(-1 + \sum_{j=0..N_i-1} \frac{N_i!}{(N_i-j)!\, j!} \right) \times M$$

where N_i is the number of sinks in the i^{th} dependency class. The first term, 1, counts the $\emptyset$ symbol, the addition of M counts the number of modes which can make a commitment to the sink (assuming every mode can make such an output commitment). Ideally, the summation should be from 1 to N-1, but this would necessitate the N=1 case being treated separately. To avoid this, the j=0 case is also included, which happens always to equate to 1, and this offset is removed by the inclusion of the "-1" term. This sum counts the different combinations in which sinks in the class can be written, and this is multiplied by the number of modes in which these outputs can occur. Obviously, the equation simplifies to:

$$\alpha S_i = \quad 1 + \left(\sum_{j=0..N_i-1} \frac{N_i!}{(N_i-j)!\, j!} \right) \times M$$

The ability to map an STS into an LTS supports the contention that STSs have the same expressive power as LTSs. The mapping has not been formalised, but sufficient explanation has been given to demonstrate that the process could automated.

6.5. STSs and Process-Based Designs

The mode-machine specification of the missile system given in Figure 6.2 can now be repeated using an STS. This is done in Figure 6.12.

It is interesting to consider an informal mapping between this STS and the proposed MASCOT design given in Figure 6.3, and the implications of the semantics of STSs on the

design. For example, as with the mode-machine, the current mode of the STS would be stored in P1; the mode transitions of the STS would be stored in activity A0; the UI sink would correspond to the UI-2 pool; the L, M and R sinks would be mapped to the single LMR-3 pool. An output commitment to a class of sinks, for example, L and M, would correspond to A2, the activity responsible for producing outputs to L and M, reading the current mode value stored in the pool P1. The fact that an STS can change mode before any of the outputs directed to the sinks to which commitment has been made can occur, is similar to the A0 activity changing the value in P1 before A2 can finish computing the outputs to send to LMR-3. The algorithm in A2 will not include another read of the mode P1, and so outputs for LMR-3 appropriate to an old mode will be produced. Consideration of pre-emptive scheduling or distributed implementations of the design reveals that this is likely to be common behaviour for such a design. A Hoare trace model of the MASCOT design would also capture such behaviours of the design; for example, it will include the trace where A2 made extremely slow progress after reading the pool. The existence of such traces is identical to the arbitrary delays in the occurrence of outputs that STS semantics allow.

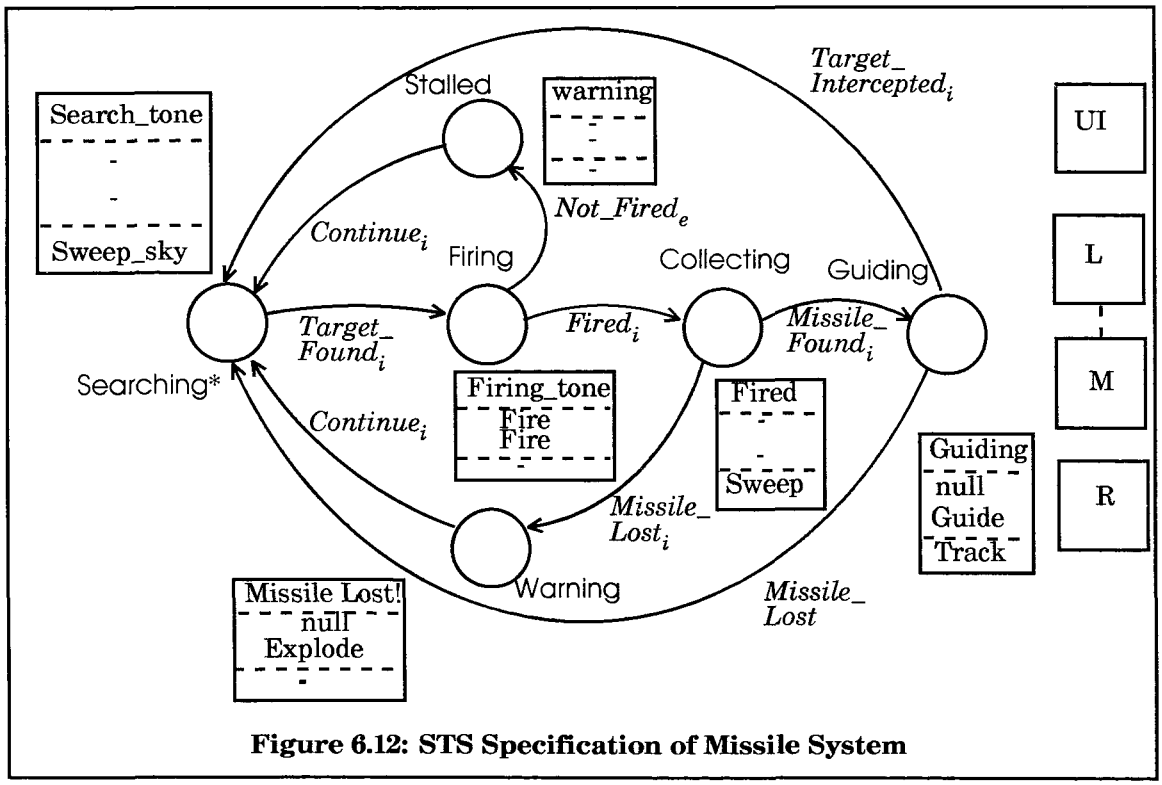


Figure 6.12: STS Specification of Missile System

The implications for the MASCOT design of an STS defining two sinks as being dependent, is that a single activity must be responsible for writing all the outputs to the IDAs which model those sinks. Certainly with Hoare Trace semantics of MASCOT, two activities cannot be guaranteed to keep in step (without explicit synchronisation messages being sent between them). Similar consideration of pre-emptive scheduling, and asynchronous distributed implementations reveals that two unconnected activities cannot be guaranteed to have the same view of the mode, and so drive the devices consistently. Thus the number of dependency classes of sinks in an STS is an effective bound on the number of activities which can be engaged in writing to output IDAs. In the missile example, the MASCOT design does not exploit the full parallelism allowed for in the STS specification, in that it uses A2 to produce outputs for R as well as L and M, thus treating the radar as though it were a critical resource with the launcher and missile.

STSs use single output actions to represent the output sent to a sink in a given mode. The corresponding MASCOT design however, has activities which change their output producing processing depending upon the mode, see Figures 6.4 and 6.5. The MASCOT design is therefore not limited to sending a single value to an IDA/sink per mode. This is a case where some form of action refinement between an STS specification and a MASCOT design would be desirable.

The informal relationship between STSs and MASCOT designs described in this section provides motivation for some of the features of STSs, such as output commitments, and helps to provide confidence that STSs have captured many of the informal intuitions which underlie the use of mode-machine as specifications of process-based designs. Formalisation of this relationship is discussed in section 6.6.4.

6.6. Refinement of STSs

Refinement is usually defined as a semantic relation, that is, as a relationship between the semantic domains of two expressions. For example, the semantic model of a design might be the set of behaviours of an implementation, and the semantic model for the design's specification might be either a (larger) set of behaviours, or a set of designs. Refinement may be defined as "set inclusion" in the first case, and "set membership" in the second.

However, while refinement is defined semantically, it is normally proved syntactically. This is because of the difficulty of reasoning about semantic models, and because syntactic

refinement can be compositional. Compositional proof systems are desirable as they enable the refinement of a compound statement to be proven with reference only to the specification of its constituent parts, rather than to the full detail of their implementations, [Hoo91]. Reasoning about semantic models usually involves reasoning about the full model, and so is not compositional.

Syntactic rules for refinement form a "calculus" or logical system for proving refinement. These rules need to be shown to be sound with respect to the definition of refinement over the semantic models. This ensures that no expression can be proven to refine another unless it actually is a refinement of it. It is also desirable for the syntactic rules for refinement to be complete. This ensures that any expression which, semantically, is a refinement of another can be proven to be so, using only the syntactic rules.

The STS semantic model is a set of valid behaviours. Therefore, refinement is a task of demonstrating that the set of behaviours of the refined STS is a subset of the set of behaviours of the original STS. Should refinement between STSs and formal MASCOT be defined, this will also involve demonstrating set inclusion, as traces semantics are also sets of behaviours.

The fact that one STS, sts2, is a refinement of another, sts1, will be written as:

sts1 $\sqsubseteq$ sts2. This is the usual refinement symbol.

Formally, sts1 $\sqsubseteq$ sts2 $\Leftrightarrow$ Obs_Beh(sts2) $\subseteq$ Obs_Beh(sts1).

Josephs, in [Jos88], has defined what it means for one state-machine to be a refinement of another by relating state-machines to processes, and by defining "refinement" between processes. Josephs identifies three basic conditions which are sufficient for state-machine refinement: one machine is a strengthening of the other; the existence of a downward simulation between the machines; or the existence of an upward simulation between the machines. Separately these rules are sufficient for refinement, together they are complete. They have been proven to form the basis of a sound and complete proof method for demonstrating refinement between state-machines. He, [He89], has used the same concepts to define process refinement rules, and Woodcock and Morgan have used them with CSP failure-divergences and action systems, [WoM90]. The approach of demonstrating refinement using upward and downward simulation was initially proposed in the context of data refinement, [HHS86].

Josephs' rules could be used directly after the STS was converted to an LTS. However, due to explosion of the state-space that this conversion results in, this approach is not advocated for any but trivial STSs. Instead, Josephs' rules are modified to apply directly to STSs. The definitions of strengthening, and upward and downward simulation need to be modified slightly before they can be applied to STSs, because of the extra elements of the STS tuple. Nevertheless, the same simulation concepts will form the basis of a proof system for STSs.

Sink and Action refinement is considered in sections 6.6.2 and 6.6.3. Refinement between STSs and MASCOT is discussed in section 6.6.4. The soundness of the refinement rules is argued in Appendix Four.

A useful function on an STS is: Next_Actions : $M \rightarrow \wp(A)$. It defines the set of actions that may occur at a given mode. A similar function is defined in [Jos88] for state-machines.

Formally:

Next_Actions(m : M) next_a : $\wp(A)$

post

$$next\_a = \{ a \in A \mid \exists m' \in M \bullet m \rightarrow^{a} m' \}$$

Next_Actions will be used in the definitions of refinement that follow.

6.6.1. Mode/Transition Refinement

In this section three rules for the mode and transition refinement of STSs are described, known as: Strengthening an STS; Downward Simulation; and Upward Simulation. The Strengthening rule allows an STS to be refined by one with the same modes, but which is more deterministic. The Downward Simulation rule enables an STS to be refined with one with more modes, providing the extra modes can be related to the modes of original STS while providing similar inter-connectivity. The Upward Simulation rule enables an STS to be refined to an STS with less modes.

6.6.1.a. Rule 1: Strengthening an STS

Given that: $sts1 = < M1, M_0 1, A1, T1, S1, \delta 1, D1 >$

and $sts2 = < M2, M_0 2, A2, T2, S2, \delta 2, D2 >$

sts1 $\sqsubseteq$ sts2 if

1. M1 = M2

2. $M_0 2 \subseteq M_0 1$

3. A1 = A2

4a. $\forall$ m $\in$ M1 • $\text{Next\_Actions}_{sts1}(m) = \text{Next\_Actions}_{sts2}(m)$

4b. T2 $\subseteq$ T1

5. S1 = S2

6. $\delta 1 = \delta 2$

7. $\forall$ t $\in$ T1 • t $\in$ dom D2 $\Rightarrow$ D1(t) = D2(t)

This asserts that one STS refines another if it is more deterministic, while responding to the same range of inputs. The STSs must have the same modes, actions, and sinks. The fact that the actions are the same for both STS, namely, A1 = A2, is intended to convey that the actions retain their same classification as inputs, outputs or events. That is: Ip1 = Ip2; Op1 = Op2; and Ev1 = Ev2.

An example of STS strengthening is given in Figure 6.13, where one of the λ labelled transitions from mode A is removed in the refinement.

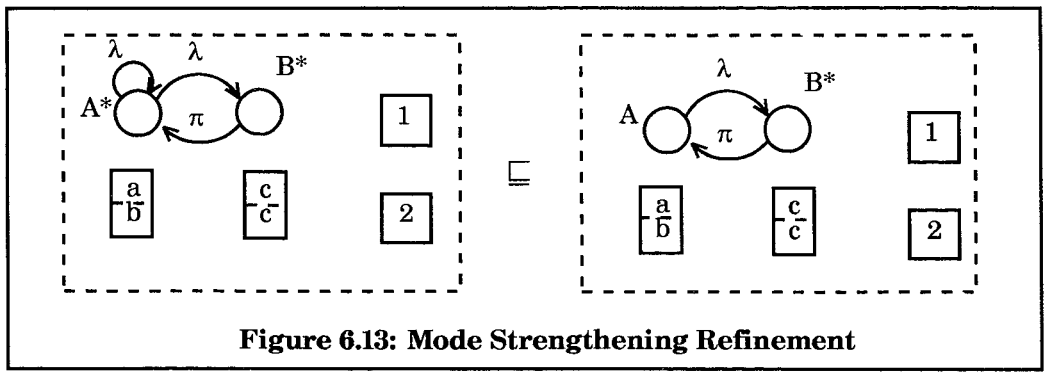


Figure 6.13: Mode Strengthening Refinement

6.6.1.b. Rule 2: Downward Simulation

One STS may refine another while introducing more modes. To establish the validity of refinement in this situation it is necessary to to demonstrate that the larger number of modes can be partitioned into an equivalence class of modes with a

one-to-one mapping with the modes of the first STS. The behaviour of the second STS, viewed in terms of these clumped modes, must be the behaviour of the first STS. The clumping is achieved by defining a relation between the modes of the two STSs, called DS, that is: $DS \subseteq M1 \times M2$, where:

$sts1 = < M1, M_0 1, A1, T1, S1, \delta 1, D1 >$ and

$sts2 = < M2, M_0 2, A2, T2, S2, \delta 2, D2 >$.

$sts1 \sqsubseteq sts2$ if

1. $\forall m \in M1, m' \in M2 \bullet$
   $DS(m, m') \Rightarrow Next\_Actions_{sts1}(m) = Next\_Actions_{sts2}(m')$

2. $\forall m' \in M_0 2 \bullet \exists m \in M_0 1 \bullet DS(m, m')$

3. $A1 = A2$.

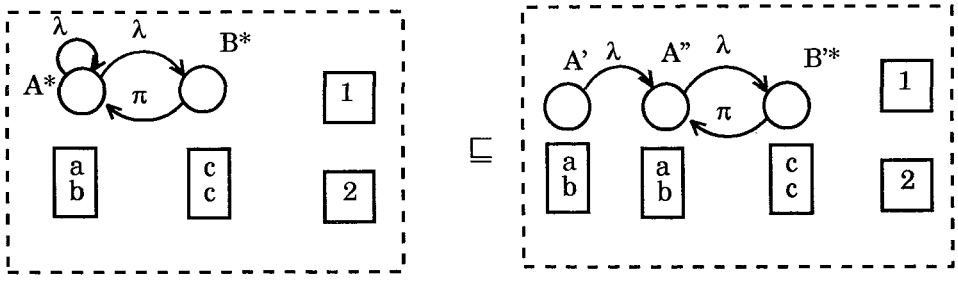
4. $\forall m \in M1, m', m'' \in M2, a \in A1 \bullet$
   $DS(m, m') \wedge m' \rightarrow^a m'' \Rightarrow$
   $\exists m''' \in M1 \bullet m \rightarrow^a m''' \wedge DS(m''', m'')$

5. $S1 = S2$

6. $\delta 1 = \delta 2$

7. $\forall t \in T1 \bullet t \in \text{dom } D2 \Rightarrow D1(t) = D2(t)$

An example of downward simulation refinement for an STS is given in Figure 6.14, where mode A* is divided into two modes, A' and A", neither of which are a possible initial state. The transitions which started from A* are distributed over A' and A".



where DS = { (A, A'), (A, A"), (B, B') }

Figure 6.14: Mode Downward Simulation Refinement

6.6.1.c. Rule 3: Upward Simulation

One STS may refine another while merging modes. To establish the validity of refinement in this situation it is necessary to to demonstrate that the larger number of modes in the first STS can be partitioned into an equivalence class of modes with a one-to-one mapping with the modes of the second STS. The behaviour of the second STS, must be at least as responsive to inputs as the first STS. The definition of which modes are merged in the refinement is given by a relation between the modes of the two STSs, called US, that is: $US \subseteq M1 \times M2$, where:

$sts1 = < M1, M_0 1, A1, T1, S1, \delta 1, D1 >$ and

$sts2 = < M2, M_0 2, A2, T2, S2, \delta 2, D2 >$.

$S1 \sqsubseteq S2$ if

1. $\forall m' \in M2 \bullet$
   $\exists m \in M1 \bullet US(m, m') \wedge Next\_Actions_{sts1}(m) \subseteq Next\_Actions_{sts2}(m')$

2. $\forall m \in M1, m' \in M_0 2 \bullet US(m, m') \Rightarrow m \in M_0 1$

3. $A1 = A2$

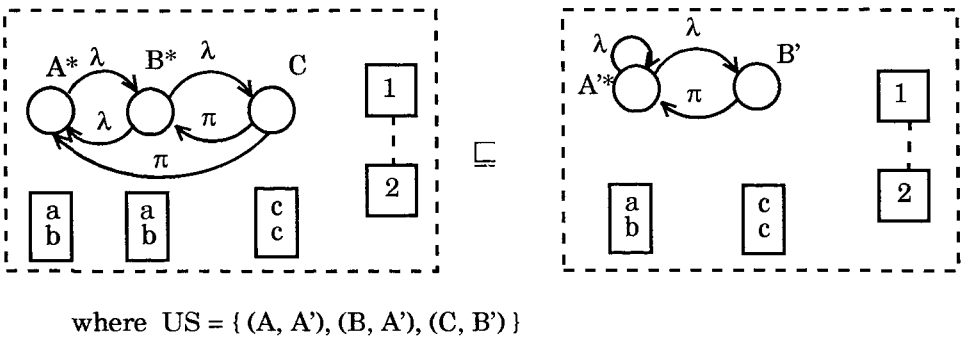
4. $\forall m''' \in M1, m', m'' \in M2, a \in A \bullet$
   $US(m''', m'') \wedge m' \rightarrow^a m'' \Rightarrow$
   $\exists m \in M1 \bullet m \rightarrow^a m''' \wedge US(m, m')$

5. $S1 = S2$

6. $\delta 1 = \delta 2$

7. $\forall t \in T1 \bullet t \in \text{dom } D2 \Rightarrow D1(t) = D2(t)$

An example of mode upward refinement is given in Figure 6.15, where two modes, A* and B*, are combined into one mode A'* during the refinement.



where US = { (A, A'), (B, A'), (C, B') }

Figure 6.15: Mode Upward Simulation Refinement

It should be noted that Rule 1 is a special case of both Rule 2 and Rule 3, where DS and US are the identity relation.

6.6.2. Sink Refinement

In the rules for refinement given so far, the STSs have had to have the same sinks, and the same sink dependencies. In this section refinements which either reduce or increase the number of sinks, or which reduce or increase the number of sink dependencies, are discussed.

The number of dependencies cannot be reduced as this would enable the refined STS to exhibit a wider range of behaviour, and so violate the definition of refinement. For example, behaviours which only produced outputs for the sinks still in the same class would be valid, while before such a "refinement" they would not. Unfortunately, neither can the number of dependencies be increased, as this also increases the number of behaviours: for example, after such a "refinement", following an output commitment to the larger class of sinks, there are more outputs which can occur.

It is also undesirable to increase the number of sinks during refinement. The problem with increasing the number of sinks is that, without action refinement, the output that was directed to one sink, must be directed towards one of the collection of sinks into which the sink has been refined.[35] This leaves the other refined sinks without an output directed to them, that is, with an STS which is not Output Complete. However, if other sinks are dependent upon one of these new sinks with no output directed to them, once one sink in the dependency class has been written to, that whole class of dependent sinks will refuse further outputs until they have been updated "consistently". However, this can never happen as, in that mode, there are no outputs directed to certain sinks in that class. This does not violate the definition of refinement, but it is not the intuitively desirable behaviour of such a refinement.

Lastly, Sink refinement which decreases the number of sinks will also not be defined. Consider, for example, the removal of one sink from a dependency class during a refinement. This will require a similar removal of all output transitions which are directed to that sink; one such transition for each mode. If the STS is not Output Complete because, in one mode, there is a sink to which it does not direct an output, and if that mode is removed during refinement, the refined STS becomes Output Complete. However, an Output Complete STS will have a wider range of behaviours than a similar one which is not, and hence this violates the definition of refinement. If, however, only Output Complete STSs are considered, there is no such problem with decreasing the number of sinks during refinement. Nevertheless, this constraint will not be imposed, primarily because the need to reduce the number of sinks during refinement is unlikely to occur.

It is concluded that there is not a sensible rule to be defined for an STS which modifies its sinks and their dependencies, and which also refines the STS.

[35] Another option must also be considered, namely, that of multiplying the transitions labelled with the output value, so all the refined sinks are written to. However, this cannot be done, due to D being a function not a relation. In any case, it violates the definition of refinement by introducing histories which contain a sequence of outputs, where, before the "refinement", only one output could occur.

6.6.3. Action Refinement

Action refinement is a highly desirable property of most refinement calculi. Unfortunately it is very problematic in concurrent languages with an interleaved model of concurrency. Aceto, [Ace92] provides the following example which illustrates this:

(1) $a \mid\mid b = (a \rightarrow b) \mid (b \rightarrow a)$

(2) $a \sqsubseteq c \rightarrow d$

but replacing the right hand side of (2) for "a" in (1) gives

(3) $(c \rightarrow d) \mid\mid b \neq (c \rightarrow d \rightarrow b) \mid (b \rightarrow c \rightarrow d)$[36].

(1) is the standard identity that holds for all interleaved models of concurrency; (2) is the basic action refinement assertion; and (3) demonstrates that the interleaving assumption is in conflict with action refinement.

A notation with action refinement needs a richer semantic model than the interleaving one, such as, for example, a partial-ordered model, [Ace92]. However, it has been argued that STSs are essentially LTSs, and hence are an interleaved model of concurrency. It should be noted that, for example, STSs contain nothing equivalent to the independency relation, ι, used in LATSs, which make LATSs partially ordered models. Therefore STS are not a suitable formalism on which to define action refinement.

The two main implications of this are that unwanted level of detail may have to be included in STSs specifications, and that the intuitive relationship between STS output actions, and the processes they are intended to represent in a process-based design cannot be formalised. This is discussed further in the following section which discusses the formal relationship between STSs and formal MASCOT designs.

6.6.4. Refinement Between STSs and MASCOT

The informal relationship between an STS specification and a MASCOT design was described in Section 6.5. Unfortunately, however, it has proven to be extremely hard to

[36] For convenience (1) and (2) have not been expressed in legal CSP, which requires the second parameter of the "$\rightarrow$" operator to be a process and not an action.

formalise this relationship, despite the clear intuitions which underlie it, and despite informal mode-based models regularly being used as a specification technique for such systems.

It is hard to define the operating "modes" of an arbitrary MASCOT design. "Mode" in a process-based design may be viewed as values stored in the system which influence the processing that the system is performing, but which are not data which are consumed during the processing. Such values may, however, be distributed around the system. They need not necessarily be stored in a separate IDA, but may be stored in variables within the activities. Things may be further confused by designs where not all the activities are aware of the same modes. This difficulty in recognising the modes of a MASCOT design is the root of the difficulty in formalising the intuitive relationship between STSs and MASCOT. The definition of a mapping between the modes of an STS and the operating modes of a MASCOT design would be the basis of a demonstration that the concurrent nature of the design does not introduce freedoms in transition behaviour which not allowed by the semantics of the STS.

An alternative approach to formalising the relationship between STSs and MASCOT designs is to relate them as state-machines. A MASCOT design could be viewed as a massive state-machine, whose state-space is the cross-product of the states of the elements of the design. Indeed, Josephs, in [Jos88], has defined the construction of a state-machine which corresponds to an arbitrary CSP program. The construction of state-machines (LTSs) from STSs has been sketched in section 6.4, and this could be formalised. Verification of a MASCOT design against an STS specification could then be performed using the simulation rules given in [Jos88]. Unfortunately, the major problem with this approach is the large state-spaces which would have to be considered in constructing the simulation relations. This is well illustrated by consideration of the size of the state-space of the missile system example used in Section 6.5. Now while the STS only has six modes, and four sinks, and is of manageable complexity, the equation defined in section 6.4 gives the size of the state-space of the corresponding LTS as being 6×7×19×7, or 5586. It is contended that this is an unmanageable size when trying to define upward or downward simulation relations. The size of the state-space of the state-machine corresponding to the MASCOT design, given by the cross-product of the state-spaces of the activities and IDAs is likely to be of comparable size, if not larger. However, increased processor speeds, and the development of efficient algorithms, means that tool-supported model-checking of state-machine refinement is becoming

possible, [BrJ92] and [FS93], and so this may not remain a totally unfruitful way of relating STSs and formal MASCOT.

A third way of formally relating STSs and concurrent designs is to relate their behaviours. An STS specification defines a set of valid behaviours for the system. As seen in earlier chapters, MASCOT designs can also be given semantics in terms of a set of valid behaviours, that is, Hoare Traces. The verification task is one of showing that all the behaviours of the design are behaviours of the specification. This may involve a mapping between the names of actions in the design, and their corresponding names in the specification. This is similar to the second approach mentioned above, but does not require the large state-space to be considered explicitly. However, it does run into the problem of the complexity of the predicates which describe the set of behaviours of the STS and design. Again this approach would fail to exploit the informal intuitions which relate mode-machines and process-based designs.

The difficulty in formalising the relationship between MASCOT and STSs was not anticipated until after STSs had been developed. It is hoped that further research will lead to a way of formalising the intuitive relationship, as it is believed the complexity of the other approaches means that they are unworkable on realistically sized systems. The solution probably lies in the consideration of a suitable subset of MASCOT designs whose modal operation is easily identifiable.

The whole issue of formalising the relationship between STSs and MASCOT is also complicated by the intended action refinement between output actions, and output producing processes. It should be possible to solve this problem by adopting richer semantic domains for STSs and formal MASCOT. Alternatively, a significant modification to STSs, which allowed processes, and not just an action, to be associated with a sink in a mode, is another approach which ought to solve this problem.

6.7. Conclusions

Formal models for describing reactive systems, called Specification Transition Systems (STSs) have been defined. STSs enable the general modes of operation of a system to be described, thereby enabling STSs to describe large LTSs compactly. A refinement calculus for STSs based on upward and downward simulation, [Jos88], has been defined. Although the intuitions for STSs were drawn from the informal use of mode-machines to specify process-based designs, a formal mapping between STSs and formal MASCOT was not

defined, although it was pointed out that they could be formally related by being mapped into LTSs. A graphical presentation of STSs was described.

Chapter 7: Conclusions and Further Work

This chapter contains a brief summary of the thesis, and a critical analysis of the value of the formal techniques which have been developed. The chapter closes with a discussion of further work.

7.1. Summary

In this thesis formal techniques have been developed with graphical presentation notations. In particular, MASCOT designs have been given a denotational semantics, and mode-based specifications for reactive systems have been proposed.

Graph grammars have been used to define the abstract syntaxes of subsets of MASCOT designs. Two increasingly complex grammars were defined, SLM_GG and BLM_GG. BLM_GG was a compact twelve rule grammar whose language corresponded to the class of MASCOT designs with loops, arbitrary branching of paths on activities, and single reader and writer IDAs. The grammar was ambiguous, but was suitable as a definition of the abstract syntax of an interesting class of MASCOT designs. The BLM graph grammar was used to structure the definition of a denotational semantics of MASCOT. Hoare Trace semantics, defined via CSP, was the model adopted.

Specification Transition Systems (STSs), a mode-based specification model, has been defined for specifying the safety properties of reactive, systems. A behavioural semantics was defined for STSs in terms of allowable transition histories. A calculus was defined for refining STS specifications.

The STS specification includes an output sink extension, for modelling the devices in a system's environment. A dependency relation between sinks allows the fact that certain devices need to be driven with sets of consistent data to be captured. These extensions reduce the number of states which need to be explicitly described, and so the number of modes which need to be explicitly mentioned in an STS specification is usually significantly less that the number of states in a corresponding state-machine. This has been demonstrated by sketching the construction of an equivalent LTS from an STS. It has been contended that STSs capture and formalise the concept of mode found in informal mode-machines.

7.2. An Evaluation of Formal MASCOT

Formal MASCOT's abstract syntax and formal semantics are evaluated in this section. The first section considers the subset of MASCOT formalised and the success of using deNCE graph grammars to define the abstract syntax. The second section reviews the adequacy of the formal model, discusses it utility, and compares it with other attempts to formalise MASCOT.

7.2.1. Formal MASCOT's Abstract Syntax

It is important to structure the denotational semantics of a program or design around the syntax of the language which is used to describe the program or design. This provides confidence that all designs expressible have been given a meaning. An abstract syntax was developed for MASCOT's graphical notation so the structure of the semantic model would better reflect the structure of the design. It was believed that a denotational model which reflected this structure would be easier to reason about and would be more intuitive. The resulting model is intuitive, but it has not been compared with an equivalent model based on an abstract syntax of a textual form of a MASCOT design.

A definition of the abstract syntax of the graphical form of MASCOT required the use of graph grammars rather than string grammars. The class of grammars known as NLC graph grammars were chosen as being suitable for this purpose, and deNCE grammars in particular. These proved to be a relatively simple, powerful, and an intuitive way of defining the abstract syntax of MASCOT designs. It was particularly satisfying that such a large subset of MASCOT designs could be characterised by a grammar with only twelve production rules. The formalisation of the embedding relation in these modern graph grammars proved to be vital in including in the language of the grammar, designs with arbitrary branching on activities. Also, the edge labels of the deNCE grammars proved to be important in formalising the definition of the actions which were hidden in the denotational model of MASCOT.

The subset of MASCOT which was described by the BLM_GG deNCE graph grammar was of a significant size. The removal of templates, subsystems, servers, and access interfaces from MASCOT-3 probably did not significantly weaken the operational expressiveness of the designs considered. However, it was unfortunate that the full range of well-formed MASCOT designs defined in chapter two could not be handled. In

particular, the removal of multiple reader pools, and multiple writer channels and signals was regrettable. It is interesting that the limitation to point-to-point IDAs is similar to the restriction to point-to-point routes which occurs in DORIS, [Sim93]. This shows that, even with this limitation, the subset of designs which have been handled is not without interest.

7.2.2. Formal MASCOT's Semantics

To evaluate the semantic model used for MASCOT designs, it is necessary to consider three aspects of the situation: the adequacy of the semantics; the utility of the formalism; and a comparison of the semantics with other models.

As discussed in chapter 4, the faithfulness of a model has to be balanced against its simplicity: the simpler the model the easier it is to reason about, but the less that may be deduced from it. The traces model captures the concurrency of the activities in terms of non-determinism, and it is the most abstract of the commonly used models of concurrency. It does not capture the liveness or progress properties of MASCOT, but these are unstudied in the MASCOT literature. One of the abstractions of the model, which the literature on MASCOT explicitly fails to make, is the reduction of the accesses to an IDA to atomic, indivisible, actions. The MASCOT-3 literature discusses multiple threads of execution which can overlap on different windows of an IDA, and even on the same window. Traces are not capable of modelling the detail of this behaviour. However, traces record the non-deterministic order of these events, which is intended to model their potential concurrency. Furthermore, the MASCOT literature asserts that an implementation of an IDA must ensure that these interleaving of threads will not interact to produce inconsistent data in the IDA. The overall behaviour ought to be as if, either one or the other got in first, and was uninterrupted. Therefore, this abstraction of traces does not conflict with the informal semantics of MASCOT.

The utility of trace semantics for MASCOT has been demonstrated for an extremely small MASCOT design, however, even reasoning about the correctness of this design proved to be a complex task. It is to be expected that richer, more faithful models of MASCOT, would result in even greater complexity, and hence, arguably, less utility. The advantage of using trace semantics via CSP, is that MASCOT can benefit from the work of the established CSP research community, again hopefully maximising the utility of formal MASCOT.

There has been a previous attempt to provide a formal semantics for MASCOT, [BJP87]. This used Broy's time stamped streams, [Bro83], to define the data-flow between components of a design, and an applicative real-time language, ART, [Bro83], to define the algorithmic components, that is, the activities and the IDAs. This formal model is much richer and more faithful to actual implementations of MASCOT, but it is more complex and harder to reason about. It is used to define the semantics of a larger class of MASCOT-3 designs, including: multiple-reader pools; multiple window IDAs; multiple thread windows; IDA to IDA paths; and subsystems. However, the relationship between the semantic model and MASCOT is left informal in [BJP87], in particular, the semantics are not structured around an abstract syntax of MASCOT. [BJP87] suggests a number of tools which could be built to exploit this semantic definition, but it does not provide a calculus for reasoning about the correctness or equivalence of MASCOT designs. The problem noted in [BJP87] concerning the industrial acceptance of applicative programming in defining the components of MASCOT designs, should be avoided with the use of CSP advocated here.

One disadvantage of the CSP traces model for semantics, is that is does not form an acceptable basis to be extended for a timed model of MASCOT. Timed CSP, [DaS89], [Sch90], [Dav91], is based on assumptions which are in conflict with the asynchronous freedom that MASCOT designs retain for the implementor. In particular, it forces a minimum non-zero, delay, between actions, and this prevents a single process from having the same model as two parallel processes with instantaneous access to a shared variable. This was the essence of the validity of the CSP model of an IDA used here. Incidentally, the timed semantics of MASCOT defined in [BJP87] is also deficient for similar reasons, as noted in the conclusions of [BJP87].

7.3. An Evaluation of STS Specifications

STSs have succeeded in introducing a mode construct which abstracts a whole class of state-machine states. It has been contended that these modes are similar to the informal modes which are used in mode-machine specifications of software systems. While this contention is essentially unprovable, some evidence for this conclusion has been given through consideration of a missile system case study derived from an actual industrial application.

The relationship between STSs and LTSs was only sketched and not formalised. However, it is not anticipated that its formalisation should provide any problems. Once the relationship between STSs and LTS is formalised, STSs will effectively have two semantic definitions, and obviously, there will be an obligation to prove their equivalence.

There were problems in generalising and formalising the mapping between STSs and process-based designs. The failure to formalise the intended mapping between STSs and MASCOT leaves open an important question over the suitability of STSs for the specification of distributed systems. It has also become clear that it would have been desirable to have given STSs a semantic model which supported action refinement.

The definition of the behavioural semantics of STSs is open to the criticism that it is rather awkward; the complexity is a little disappointing, but this probably reflects the complexity of the intuitions which underlie mode-machines. The refinement calculus for STSs is similar to the one for state-machines, proposed in [Jos88]. The proof of the soundness of the refinement calculus, given in Appendix 4, means that a user of STSs may refine them without having to manipulate their semantic model.

It is obvious that STSs treat inputs and outputs asymmetrically. Input devices, or sources, are not distinguished, unlike sinks. The reason for this is that they are not needed to enable STSs to capture the essence of mode. However, it can be argued that for some systems, it would be desirable to be able to express the requirement that certain inputs must be consumed and processed as though the system were in a single mode. It can therefore be argued that STSs ought to include sources as well as sinks[37].

STSs may be sensibly compared with Statecharts and Temporal Logic, both formalisms advocated for the specification of reactive systems. Statecharts, [Har87], are a state-based graphical specification notation for reactive systems. Statecharts contain a number of features not present in STSs, for example, a hierarchical structuring mechanism, and the decomposition of states into concurrent state-machines with broadcast communication. The hierarchical presentation of Statecharts enables the number of top-level states to be kept manageable. However, these top-level states retain the properties of states, namely, that at any given time, the system is in one, and only one, such top-level state. In contrast, STSs keep the number of "states" manageable by changing their properties, and turning them into "modes". The modes of an STS are effectively a class of states. It is argued that this is a

[37] Thanks are due to Prof. John McDermid of York University for pointing out this need.

better model for reactive systems which will be implemented as distributed systems, especially distributed systems with asynchronous communication, because, with such systems, a global, synchronised, change of state cannot be ensured. Nevertheless, STSs would undoubtedly benefit from a hierarchical mechanism. The modes of an STS cannot be decomposed into parallel state-machines as can the states of a Statechart. However, as argued in [IOW90], it is debatable whether a specification language for reactive systems should explicitly specify concurrency. The position has been adopted that STSs should not. Statecharts do not have anything which corresponds to the sinks, and dependency classes of sinks of STSs.

Temporal Logic (TL) is a significantly different approach to the specification of reactive systems. A TL specification defines a set of valid models which exhibit the properties stated in the TL expression. These models can be interpreted as either a set of behaviours, or as a set of models of the system. TLs are more powerful than STSs, in that they allow discussion of infinite behaviours, and hence the expression of fairness requirements. However, as has already been discussed, this is not relevant in the specific domain of STSs and MASCOT designs. TLs make no commitment to state; they only record the required ordering relationships between the occurrence of events. TLs are a more flexible specification language, in that TL specifications can be extended with a new property by conjoining a new expression. STSs, in contrast, define a particular structure or model; the properties of the system are dependent upon the actual model. The need to record that the system has another property may require the entire structure to be changed. However, a well-formed STS specification always defines a system, while a well-formed TL specification may be contradictory. Furthermore, it may not be obvious that a TL specification is contradictory.

TLs have an associated calculus, so a specification in one form can be proven to be equivalent to a specification in another form. The refinement calculus plays a similar role for STSs, but again TLs are more flexible in this respect. STSs have a nice graphical presentation scheme, with natural intuitions, whereas a TL specification, is usually a dense textual expression, and the precise interactions between, and the semantics of, the logic operators carry important information. An STS specification is likely to be smaller than the TL equivalent. This is because a lot of information is stored in the precise semantics of an STS model. A TL equivalent would have to contain explicitly all this semantic information. This will significantly obscure the intended behaviour which is specific to the system being specified.

7.4. Further Work

The work described in this thesis is the result of a three year period of research. It does not claim to be a finished piece of work, in that there are some significant loose ends, and there are a number of ways in which the work presented here could be extended and improved. This section collects together a number of suggestions as to what could sensibly be done to develop the work.

7.4.1. Formal MASCOT

Chapter three contained a full discussion of different elements of formalising a design method, and it identified that only MASCOT's way of modelling would be formalised in this thesis. Obviously the other elements of MASCOT would benefit from the clarity of formalisation, in particular, its way of control and working; that is, its process model.

There is further work which should be done on formalising MASCOT's way of modelling. In particular, an unambiguous graph grammar needs to be developed for the formalisation of the concrete syntax of graphical MASCOT designs, thus enabling a unique derivation tree to be assigned to a MASCOT design. There should also be extended concrete and abstract grammars for all well-formed MASCOT designs, in particular, it would be desirable to be able to include designs with multiple reader pool IDAs, and multiple writer channels and signals. It should be proven that the language of these grammars coincides with the class of well-formed designs defined in chapter 2. The denotational model of MASCOT would need to be extended for such a larger abstract syntax. It is anticipated that a deNCE graph grammar will suffice for this task, and that the denotational model will follow the pattern already established in chapter 5 with the BLM_GG.

The BLM_GG abstract syntax provided should be suitable for structuring the definition of any richer denotational models that might be required for MASCOT. There is probably a case for special, tailored, semantic models of MASCOT for particular implementation schemes of MASCOT designs. For example, models tailored for Ada, [LRM83], Occam, [INM88], or DIA[38], [Sim91], implementations could be much stronger in their modelling of the progress, liveness, and scheduling of activities. Clearly, calculi

[38] Data Interaction Architecture

would need to be developed with these richer models to support proof of correctness. Tailored models such as these are a relatively unexplored area of concurrent formal methods.

Related to these tailored models, is the development of a general formal model for MASCOT which handles time quantitatively, yet which retains the freedom for asynchronous implementations. Currently the temporal properties of MASCOT designs is the subject of an ongoing research project, SPIRITS, [IED91].

7.4.2. Specification Transitions Systems

There are two desirable extensions to STSs that have already received some study: namely, the addition of triple column output vectors per mode, similar to the extensions for state-machines advocated by Avnur, [Avn90], and the addition of hierarchical nesting of STSs. The triple output vectors would allow the definition of the initialisation output which should be sent to each sink upon entry to a mode, the definition of the steady mode output, as currently, and the definition of a tidying up output, which occurs upon exit from the mode. The hierarchical nesting of STSs would enable the complexity of larger systems to be managed. Obviously, the refinement calculus and its soundness proofs would need to be extended to cover such extra features.

It would be desirable to formalise the construction of an LTS from an STS, and to prove that the set of behaviours of such an LTS is the same of the set of behaviours of the STS.

Perhaps the most important extension to the current theory would be the formalisation of the relationship between STSs and formal MASCOT designs. It is anticipated that such formalisation will involve defining a subset of MASCOT designs which can be related to STSs, and the use of an abstraction function which extracts "mode" information from such MASCOT designs.

It is clear that STSs and MASCOT would benefit from a richer semantic model which supported action refinement. Alternatively, it would be interesting to explore the reformulation of STSs with output processes rather than only actions. This may be a way of reducing the need for action refinement. Nevertheless, there is merit in the argument that the abstract nature of STS specifications compared with MASCOT

designs means that it is totally inappropriate to have to specify at the level of detail of the "actions" of a MASCOT activity.

Another area of further research is the extension of STSs to include the specification of quantitative temporal requirements, and the definition of their relation to a temporal MASCOT semantics. It is possible that such research will conclude that dividing sinks into dependency classes is not necessary if time is handled quantitatively. The reasoning may be that, specific constraints can be imposed on the delay between updating different sinks, replacing the need for collecting such sinks together in a dependency class. Presumably, with temporal extensions, the constraint that a single activity would have to look after all the sinks in a class, could be loosened to allow multiple activities to be engaged in producing the outputs, providing the activities could be demonstrated to meet the temporal requirement that all the sinks must be updated within the time bound.

APPENDIX ONE: Introduction to Formal Systems

When dealing with formal systems it is important to maintain a clear distinction between the *language* of the system or logic, and the *meta-language* used for discussing and reasoning about the system. The language of a formal system defines what constitutes a *well-formed sentence* in that system; it consists of a (possibly infinite) set of symbols, along with rules for combining these symbols into sentences, or "well-formed expressions".

The well-formedness rules may state that some symbols in the given system must take other symbols as parameters; for example, predicate symbols in first order predicate logic are normally distinguished by their arity, that is, by the number of parameters they require. It is normal for there to be a relatively simple finite algorithm for determining if a given expression is a well-formed sentence of a given formal system.

In a formal system, certain well-formed sentences are distinguished from the others by being called *axioms*. Axioms are taken to be "true" (or valid) in a sense which will defined later. Axioms form the *basis* of a formal system.

As well as a language and axioms, a formal system also consists of a (non empty) set of *inference rules*. An inference rule consists of two parts: a list of well-formed sentences known as *hypotheses*; and a well-formed sentence known as a *conclusion*. The intention of a system's inference rules is to capture all the basic reasoning steps which are valid in the logic. That is, if the hypotheses of a rule are valid or true, so must the conclusion to the rule. Axioms may be presented as inference rules with a null list of hypotheses.

A *theorem* of a formal system is a well-formed sentence in the language and must be the conclusion of an inference rule for which every hypothesis of the rule is either an axiom or another theorem of the system. In other words, a sentence is only a theorem when it is well-formed, and there exists at least one sequence of inference rules from the axioms of the system to that theorem. The fact that an expression, E, is a theorem of a formal system, F, is sometimes written, $\vdash_F E$. An expression which can be proven, given the assumption of extra hypotheses, H, is written $H \vdash_F E$.

So far the truth or validity of a sentence has only been discussed informally. Actually these concepts can only be formalised with respect to an *interpretation* of a formal system[39].

[39]There can also be an informal notion of extra-systemic validity, however this is not under-consideration here. Further discussion can be found in [Haa78].

A formal system as described so far, is simply a set of rules for manipulating sentences composed of meaningless symbols. However, symbols, axioms, and inference rules are usually chosen to reflect some meaning. This concept of meaning is formalised by the interpretation of the well-formed sentences of the formal system in some mathematical structure. An *interpretation* maps well-formed sentences to members of a given mathematical *structure*.

Certain members of the mathematical structure are designated to be *truthful*. If all the axioms of a formal system map by the interpretation to members of structure which are designated as "truthful", the structure is said to be a *model* of the formal system. It is in this sense that axioms are said to be "true". Clearly, a formal system may be interpreted in numerous different models. A model may be said to define (one) semantics of the formal system. A formal system can be seen as defining a set of models; a set of possible semantics.

The fact that an expression E is true when interpreted in a model, M, of formal system F, is written $M \models_F E$.

A formal system for which every theorem maps to a true member of a given model of the system is said to be *sound,* or *consistent with respect to that model and interpretation*. This can be expressed, as, a system, F, is sound with respect to a model M, if, and only if,

$$\forall E \bullet \vdash_F E \Rightarrow M \models_F E.$$

A sound formal system is one where the inference rules capture valid reasoning with respect to a given interpretation and underlying mathematical structure. A formal system for which no sentence can be proven to be both true and false is said to be *consistent*, or *consistent with respect to provability*.

A formal system for which there exist a theorem for every member of a model which has been designated truthful is said to be *complete* with respect to that model. This can be expressed, as, a system, F, is complete with respect to a model M, if, and only if,

$$\forall E \bullet M \models_F E \Rightarrow \vdash_F E.$$

Such a formal system can be used to prove any "truth" that holds for the members of its model.

Obviously, it is desirable to deal with a sound and complete formal system and model whenever possible. This ensures that if something is true it can be proven, and if something is proven it will be true. Unfortunately, Gödel has demonstrated that any formal system capable of capturing the reasoning of arithmetic is so expressive that it can express a sentence which asserts that if the system is consistent the sentence can neither be proven nor dis-proven, [Göd31]. This means that formal systems that can express arithmetic cannot be complete.

APPENDIX TWO: Reactive Systems

Reactive systems are an important class of systems which are built by engineers; many software controlled systems are reactive. However, many slightly variant definitions of reactive systems have been given, as can be seen in: [Led90]; [AlR91]; and [Bro91]. One of the earliest definitions is by Pnueli:

> *"Reactive systems are systems that cannot adequately be described by the relational or functional view. .... Typically, the main role of reactive systems is to maintain an interaction with their environment, and therefore must be described (and specified) by their on-going behaviour"*, [Pnu86b].

The common element to the various definitions of reactive systems is the mention of the system's environment. This is significant, as the "interaction" of a reactive system with its environment depends upon the system and its environment being able to engage in autonomous actions. The system and its environment must be viewed as being concurrent. This point is made in the following quotation:

"A fundamental element in reactive programs is that of concurrency. By definition, a reactive program runs concurrently with its environment", [MaP92].

This concurrency is the distinguishing property of a reactive system. It is not distinguished by non-termination: a reactive system may terminate after having maintained some interaction for a period; nor is it distinguished by internal concurrency: an internally concurrent system may calculate a function, and then terminate, and its environment may do nothing of importance until the result is calculated[40]. Reactive systems do not necessarily have to be real-time systems in the sense of being dependent upon the quantified timing of inputs. Reactive systems may simply be dependent on the relative ordering of inputs with respect to system events.

The STS model described in this thesis has been proposed as a specification notation for reactive systems. It should be noted that finite state machines (FSMs) are not an appropriate specification notation for reactive systems. The reason for this is that FSMs implicitly assumes that the inputs from the environment, and the transitions of the machine, take turns: and hence there is no concurrency between the machine and its

[40]Manna and Pnueli make the point in [MaP92] that the internal processes of a concurrent system which is non-reactive, must be understood as reactive systems. The other processes act as the concurrent environment for each other.

environment. This limitation holds in spite of the standard automata theoretic result that finite state machines with ε–moves have the same expressive power as finite state machines without ε–moves, [HoU79]. An FSM which can engage in autonomous transitions (ε–moves) is a reactive system. However, the equivalence between FSM with ε–moves and those without, is one of language recognition, and not one of on-going behaviour. The need for a stronger equivalence than the one used in automata theory is argued in Chapter 4.1 of [Mil89].

LTSs, however, are suitable for specifying reactive systems as they can contain transitions labelled with actions which are not inputs, and so can engage in internal behaviour independent of, or concurrently with, their environments. It is thus significant that STSs are based on LTSs and not FSMs.

APPENDIX THREE: Formal Models of Concurrency

In this section the various formal models discussed in chapter four are defined.

Labelled Transition Systems

Labelled Transition Systems (LTSs) are interleaved state-based models. They are defined by the tuple: <S, s, A, T>, where S is a set of states; s is the set of initial states, $s \in S$; A is a set of actions or events; and T is transition relation: $T : S \times A \rightarrow \wp(S)$, which is a function from states and actions to a set of successor states, each of which must be in S. If the set is null, then the transition is not enabled on that state. (Sometimes slightly variant definitions are given: for example, [Hen88], [Mil89], and [Nie92], but they are all essentially the same.)

It should be noted that LTS are not the same as finite-state-machines, (FSMs). An FSM only changes state upon receipt of an input, while an LTS simply records the events that occur on transitions between states, that is, by the action label on the transition. LTSs are therefore suitable for modelling reactive systems because autonomous transitions are simply described by labelling transitions with non-input actions.

Concurrency is present in LTSs only as an interpretation of non-determinism. A particular state may have more than one possible transition defined from it, these may be taken non-deterministically. This may be viewed as occurring "concurrently" using the interleaved model of concurrency. Although this is not a particularly intuitive model of concurrency, it often forms an adequate underlying model, for some more intuitive notation, such as a process algebra, [Hen88].

Using the notation of $s_1 \rightarrow^a s_2$ to mean $(s_1, a, s_2) \in S$, and $s_1 \rightarrow^v s_n$, where v is a sequence of actions, $a_1, a_2, a_3 \ldots a_n$, to mean that there is a sequence of states, $s_1, s_2, s_3, \ldots s_n$, such that, for all $i \in 1..n-1$, $s_i \rightarrow^{ai} s_{i+1}$, it is possible to define the following important properties of an LTS:

A state, s', is *reachable* iff there exists a v, such that, $s \rightarrow^v s'$.

An LTS is *acyclic* iff for every state, s', there exists a v, such that, $s' \rightarrow^v s'$.

Synchronisation Trees

Synchronisation Trees (STs) are interleaved, branching history-based models of concurrency. They are basically trees, where the nodes represent program states, and the branches from each node are the various transitions from that state. The branches are labelled with the actions involved in that transition. They are also called derivation trees in [Mil89]. STs are basically just the unfolding of LTSs. A state which is visited more than once in a given history will give rise to a different node in the ST each time that it is visited; however, STs fail to record the identity of these nodes.

An ST may be viewed as an acyclic LTS where every state is reachable from the initial state, and each state has a unique predecessor. Formally, every state having a unique predecessor may be defined as:

$s' \rightarrow^{a} s'' \wedge s''' \rightarrow^{b} s'' \Rightarrow a = b \wedge s' = s'''$, for every state, s', s", and s'".

Hoare Traces

Hoare Traces (HTs) are interleaved, linear history-based models of concurrency. They are sets of linear histories of system actions. They satisfy the properties that a null history is always present, and that the set of traces is always prefixed closed. That is, if one history is a possible behaviour of a system, so is any prefix of that history.

The concurrent occurrence of two events is captured by the fact that they will occur in both orders in different traces in the set. Obviously, this fails to distinguish between non-determinism and concurrency. HTs also fail to distinguish between a system which makes a non-deterministic choice between two executions which start with the same event, and then behaves differently, and a system which engages in the common event, and them makes a non-deterministic choice between different behaviours. This information either needs to be captured by the use of STs mentioned above, or the use of extra sets of traces, such as refusal and divergences sets, as defined for CSP, [Hoa85].

Formally, an HT = <H, A>, where H is set of sequences of actions, and A is a set of actions. $H \subseteq A^*$, which is prefixed closed:

$a_1, a_2, a_3, \ldots, a_n, a_{n+1} \in H \Rightarrow a_1, a_2, a_3, \ldots, a_n \in H$.

A^* is the set of possible sequences of the elements of A.

Labelled Asynchronous Transition Systems

Labelled Asynchronous Transition Systems (LATSs) are non-interleaved state-based models of concurrent systems. They are defined by the tuple, [Kwi89]: $<S, S_0, A, T, \iota>$, where S is a set of states; S_0 is the set of initial states; and $S_0 \subseteq S$. A is the set of actions or events. T is the transition function: $T : S \times A \rightarrow \wp(S)$, a function from states and actions to a set of successor states, each of which must be in S. If the set is null, then the transition is said to be "not enabled" on that state. ι is an independency relation between actions; $\iota : A \rightarrow A$. ι is a symmetric and irreflective relation on actions. Actions which are independent may execute concurrently, while those which are not are considered to be causally related.

LATS are significantly different to LTS described above, for example, they distinguish concurrent actions from non-deterministic choice of actions. Also, while retaining a state concept, an LATS can effectively be in more than one state during an execution. Like LTSs, LATSs are suitable for modelling reactive programs because the actions labelling each transition need not include an input event, and hence autonomous actions are describable.

Unfortunately, LATS do not form a natural semantics for process algebras. Process algebra usually identify processes as being concurrent, while LATS identify actions. However, actions in a process algebra may alternate between being concurrent and not. Consider, for example the CSP process:

$a \rightarrow b \rightarrow (a \rightarrow P \mid\mid b \rightarrow Q)$.

A similar point is noted in [Kwi91], but this is not considered an insurmountable problem. Indeed, by mapping actions which are dependent anywhere in an expression, to dependent actions, Kwiatowska argues the desired semantics are obtained for synchronous communication.

Labelled Event Structures

Labelled Event Structures (LESs) are non-interleaving. branching. history-based models of concurrent programs, [Win87]. They are effectively a set of tree histories, and they bear the same relation to LATS as STs do to LTSs. The trees record the parallel executions of LATS, the actions in each tree cannot be related to the actions in other tree, hence revealing LESs as being a form of partial order semantic model of concurrency.

The trees record the branching history of each concurrent process, and hence can distinguish between when non-deterministic choices are made.

Formally, a LES is a three-tuple, $(E, \leq, \#)$. where E is a set of events, $\leq$ is a partial order on events ($\leq \subseteq E \times E$), which relates dependent events, and # is a symmetric relation on events, ($\# \subseteq E \times E$) which defines conflict between events, and

1) $\forall e', e'', e''' \in E \bullet e \# e' \leq e'' \Rightarrow e \# e''$

2) $\forall e \in E \bullet \{ e' \mid e' \leq e \}$ is finite.

Mazurkiewicz Traces

Mazurkiewicz Traces (MTs) are non-interleaving linear history-based models of concurrent programs, [WiN92]. An MT model is a partial order of traces, where each class of traces which are related indicates the linear trace behaviour of one concurrent processes. The actions in traces which are not related by the partial ordering are unordered so as to indicate that they come from concurrent (perhaps distributed) processes.

Non-determinism is modelled by interleaving actions within one trace. MTs hence do not distinguish where a non-deterministic program makes its choices, unlike LESs. This is similar to the relationship between HTs and STs.

Formally, an MT = $< M, A, \iota >$, where M is the set of traces, A is the set of actions, and ι is a symmetric irreflective relation on actions which defines their independence. $M \subseteq A^*$ and $M \neq \emptyset$.

MTs are:

Prefixed Closed: $\forall s \in A^*, a \in A \bullet sa \in M \Rightarrow s \in M.$

I-Closed: $\forall s, t \in A^*, a, b \in A \bullet sabt \in M \wedge a \iota b \Rightarrow sbat \in M.$

Coherent: $\forall s \in A^*, a, b \in A \bullet sa \in M \wedge sb \in M \wedge a \iota b \Rightarrow sab \in M.$

APPENDIX FOUR: Soundness of the Refinement Rules

In this appendix a natural deduction style proof, [Jon90], is given of the soundness of the first STS refinement rule, and a more informal, mathematical style argument is given of the soundness of the third rule. The second rule would be proven similarly to the third. The first proof is not discharged with full formally, but most steps are justified with reference to an explicit inference rule, and the ones that are not follow from basic logic or arithmetic.

Logical Rules

The following logical, set and sequence theoretic inference rules are used explicitly in the soundness proofs that follow.

$$\text{X1}\quad \frac{a \Rightarrow c}{a \wedge b \Rightarrow c \wedge b} \qquad \text{X2}\quad \frac{(a \Rightarrow b) \wedge (c \Rightarrow d)}{((a \wedge c) \Rightarrow (b \wedge d))}$$

$$Def_{\subseteq}\quad \frac{S \subseteq T}{\forall s \bullet s \in S \Rightarrow s \in T} \qquad \text{Set}\quad \frac{\forall x \bullet P(x) \Rightarrow Q(x)}{\{ y \mid P(y) \} \subseteq \{ z \mid Q(z) \}}$$

$$\text{Seq1}\quad \frac{\forall x : X \bullet P(x)}{\begin{array}{l}\forall s : X\text{-seq} \bullet \\ \forall i \in 0 .. (\text{len } s) - 1 \bullet P(s_i)\end{array}}$$

Soundness of the STS Refinement Calculus

Three syntactic refinement rules for STSs were defined in Chapter 6. Refinement was defined in terms of the semantic behavioural models of STSs. It is necessary to prove that when any of the rules are used to demonstrate an STS refines another STS, the first actually is a refinement of the second. That is, it is necessary to prove the soundness of each of the three refinement rules. This is done by proving that one STS refines another, assuming only the conditions that each rule requires.

Thus, if a rule has seven conditions, for example, h1 to h7, it is necessary to demonstrate that:

$$h1, h2, h3, h4, h5, h6, h7 \vdash \text{Obs\_Beh}(sts2) \subseteq \text{Obs\_Beh}(sts1)$$

Soundness of Rule 1

Numbering the eight conditions of Rule 1 as h1 to h4a, and h4b, to h7, it is necessary to demonstrate:

h1, h2, h3, h4a, h4b, h5, h6, h7 $\vdash$ Obs_Beh(sts2) $\subseteq$ Obs_Beh(sts1)

The proof is:

from h1, h2, h3, h4a, h4b, h5, h6, h7

1. $\forall$ t : TRANSITION • t $\in$ T2 $\Rightarrow$ t $\in$ T1 [*h4b, Def*$_{\subseteq}$]
2. $\forall$ m • m $\in$ $M_0$2 $\Rightarrow$ m $\in$ $M_0$1 [*h2, Def*$_{\subseteq}$]
3. $\forall$ t : TRANSITION • start(t) $\in$ $M_0$2 $\Rightarrow$ start(t) $\in$ $M_0$1 [*2*]
4. $\forall$ t : TRANSITION • start(t) $\in$ $M_0$2 $\wedge$ t $\in$ T2 $\Rightarrow$ start(t) $\in$ $M_0$1 $\wedge$ t $\in$ T1 [*1, 3, X2*]
5. $\forall$ d : TRANSITION-seq •
   start(d_0) $\in$ $M_0$2 $\wedge$ $\forall$ i $\in$ 0 .. (len d) - 1 • d_i $\in$ T2 $\Rightarrow$
   start(d_0) $\in$ $M_0$1 $\wedge$ $\forall$ i $\in$ 0 .. (len d) - 1 • d_i $\in$ T1 [*4, Seq1*]
6. $\forall$ d : TRANSITION-seq •
   d $\in$ { x | start(x_0) $\in$ $M_0$2 $\wedge$ $\forall$ i $\in$ 0 .. len x - 1 • x_i $\in$ T2 } $\Rightarrow$
   d $\in$ { x | start(x_0) $\in$ $M_0$1 $\wedge$ $\forall$ i $\in$ 0 .. len x - 1 • x_i $\in$ T1 } [*5, set2*]
7. $\forall$ d : TRANSITION-seq • d $\in$ Der*(sts2) $\Rightarrow$ d $\in$ Der*(sts1) [*6, Def Der**]
8. $\forall$ d : TRANSITION-seq • d $\in$ Der*(sts2) $\wedge$ Is_Serial(d$\lceil$(Ip2 $\cup$ Ev2)) $\Rightarrow$
   d $\in$ Der*(sts1) $\wedge$ Is_Serial(d$\lceil$(Ip2 $\cup$ Ev2)) [*7, X1*]
9. $\forall$ d : TRANSITION-seq • d $\in$ Der*(sts2) $\wedge$ Is_Serial(d$\lceil$(Ip2 $\cup$ Ev2)) $\Rightarrow$
   d $\in$ Der*(sts1) $\wedge$ Is_Serial(d$\lceil$(Ip1 $\cup$ Ev1)) [*8, h3, Sub of identities*]

10: $\forall$ d : TRANSITION-seq • d $\in$ IES_Derivations(sts2) $\Rightarrow$ d $\in$ IES_Derivations(sts1)
[*9, Def IES_Derivations*]

11: $\forall$ d : TRANSITION-seq • d $\in$ IES_Derivations(sts2) $\wedge$ F(d, sts2) $\Rightarrow$
d $\in$ IES_Derivations(sts1) $\wedge$ F(d, sts2)
[*10, X1, F being an unassigned proof variable*]

12: F(d, sts2) = F(d, sts1), [*If F is only dependent on M, A, Op, OC, S, δ, D, and common transitions in d. All of which are identical for sts1 and sts2*, h1..h7]

13: $\forall$ d : TRANSITION-seq • d $\in$ IES_Derivations(sts2) $\wedge$ F(d, sts2) $\Rightarrow$
d $\in$ IES_Derivations(S1) $\wedge$ F(d, sts1) [*11, 12*]

14: ∀ d : TRANSITION-seq • STS_Der(d, sts2) ⇒ STS_Der(d, sts1)
[*13, Defn. of STS_Der, and F is defined as the second clause of the STS_Der predicate in section 6.2.2, which satisfies the conditions for step 12.*]

15: Beh(sts2) ⊆ Beh(sts1) [*14, Defn. of Beh, Set*]

infer Obs_Beh(sts2) ⊆ Obs_Beh(sts1) [Defn. of Obs_Beh]

Soundness of Rule 3

The soundness of rule 3 could be proven using a natural deduction style proof, similar to rules 1 and 2, however, a more informal "mathematical-style" argument is given instead. It is hoped that this will make the reasons for the soundness of the rule clearer by presenting the proof more abstractly.

Rule 3 is sound if it can be shown that its seven sufficient conditions for sts1 to be refined by sts2, imply that the observable behaviours of sts2 are a subset of the observable behaviours of sts1.

The observable behaviours of an STS are the sequences of actions which label derivations in the set "Beh" for that STS. A derivation is in the set "Beh" if it satisfies the predicate STS_Der. There are two conjoined clauses to STS_Der. The first clause constrains input and event labelled transitions to occur "serially" in a derivation, while the second places different constraints on all the output and output commitment labelled transitions in each derivation.

It is first proved that, when only input and events are considered, the observable behaviours of sts2 are a subset of sts1. It is then proved that this is also true when outputs are considered.

(1) The first clause of STS_Der requires that the derivations of both sts1 and sts2 will be IES_Derivations. However, sts2 may have less modes than sts1, and have different transitions in its derivations as a result. Constraint 4 on rule 3, requires that for every transition that sts2 can engage in, there is a corresponding transition in sts1, defined between modes, which are related to the modes of the sts2 transition by the US relation. US may reduce the number of modes in sts2. Therefore there may be transitions which sts2 can engage in, that sts1 cannot, due to the change of mode names associated with certain transitions. However, when observable behaviours are considered, variant modes are

abstracted from. Constraint 4 ensures that, while transitions may become defined on amalgamated modes, the actions available at each step will be the same. This, coupled with constraint 2, which ensures that the initial modes of sts2 are suitably related by US to initial modes in sts1, means that sts2 IES derivations have the same observable behaviour as sts1 IES derivations.

(2) The second clause of STS_Der constraints the occurrence of transitions labelled with outputs and output commitments in STS behaviours. Output commitments are considered first. These are constrained to be "serial" with respect to other inputs and events, being event actions. However, they are further constrained to only occur when they are directed to sinks in a class not currently committed to in the derivation. This means that either, i), a commitment has not yet been made to that class of sinks, or ii), a commitment has been made, but it has been subsequently followed by a full set of outputs to that class of sinks. Constraints 5 and 6 ensure that sts1 and sts2 share the same sinks and sink classes.

In the first case, i), sts2 will only engage in such an output commitment in series, and we know from part 1 of the proof, sts1 will be able to engage in the same sequence of event actions. So the observable behaviour will be the same. In the second case, ii), sts2 will only be able to engage in an output commitment which sts1 cannot, if sts2 is able to complete the set of outputs to the class of sinks in a way which sts1 cannot. Output actions are considered next, and it will be argued that this is will not be the case, and so the observable view of output commitments of sts2 are a subset of sts1's output commitments.

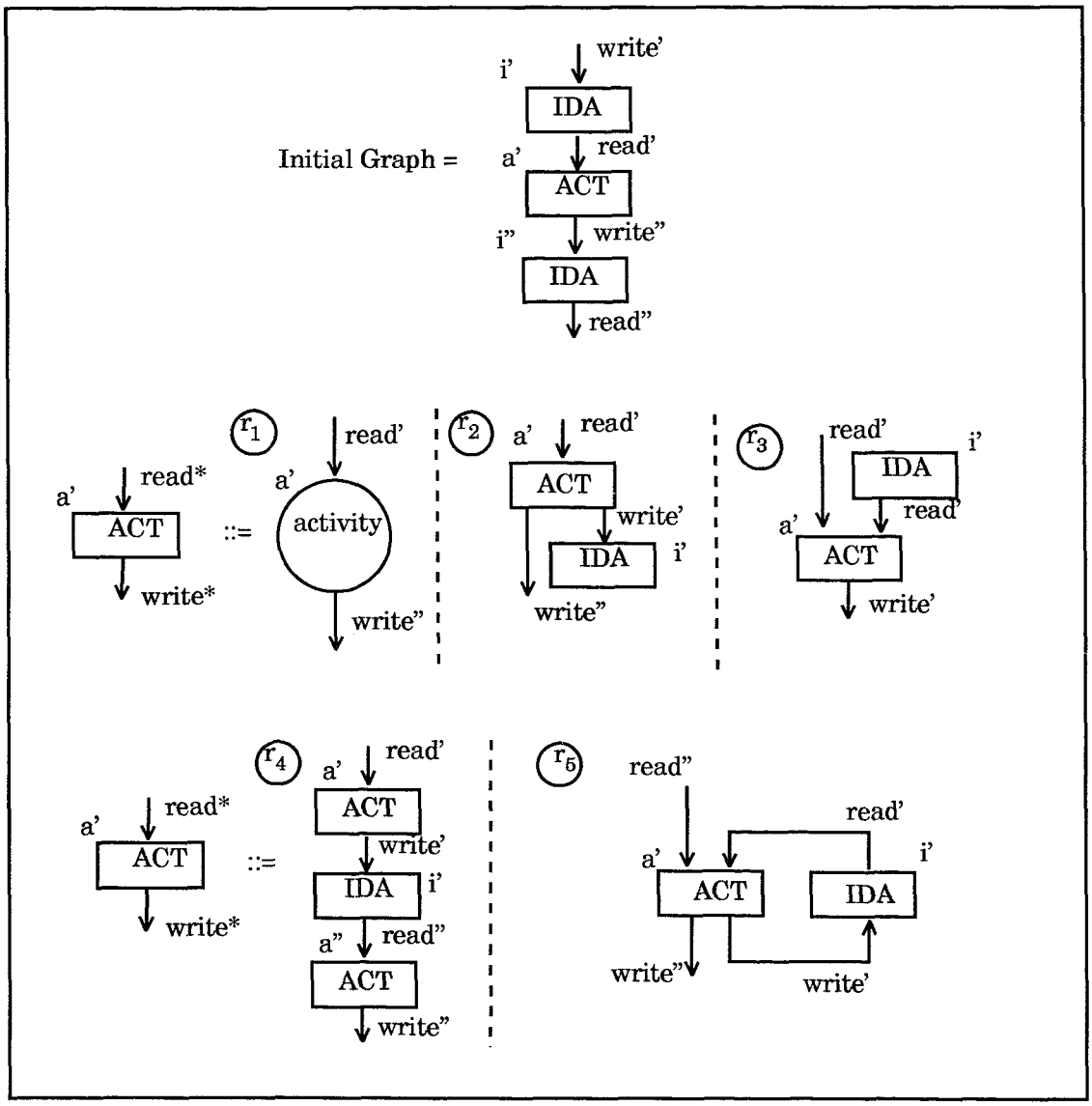
Output labelled transitions can only occur once in a derivation after an appropriate output commitment, and before a subsequent commitment. The start mode of an output labelled transition must correspond to the start mode of the transition which was labelled with the output commitment. However, sts2 may have less modes than sts1, and have different transitions in its derivations as a result. Constraint 4 requires that for every transition that sts2 can engage in, there is a corresponding transition in sts1, defined between modes, which are related to the modes of the sts2 transition by the US relation. In particular, there must be the same output labelled transitions, allowing for the amalgamation of modes allowed by the US relation. Therefore, considering only the sequences of actions which may occur, sts2 has defined the same output actions as sts1 per "equivalent" mode. Output labelled transitions, of course, need not be "serial". They may occur arbitrarily after the associated output commitment. It has already be argued that sts2's first output commitment per class is only occur to occur if sts1 can engage in the same output commitment. Therefore, in the start up case, sts2 outputs may only occur when

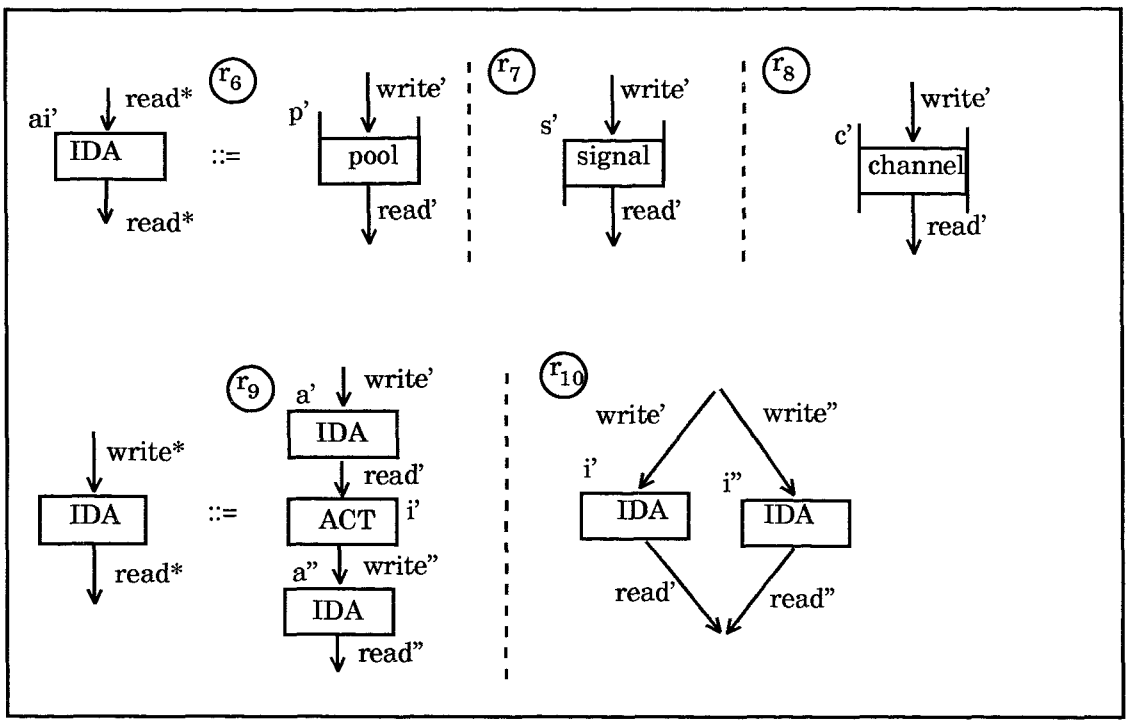
sts1's output commitments. Assuming that up to an arbitrary point in a derivation sts2 outputs are the same as sts1 outputs, the output commitments each can engage in are identical, and so the next output actions that they can engage in are identical. Using induction, it is therefore possible to conclude that all sts2 outputs only occur when sts1 outputs can occur. Therefore subsequent output commitments for sts2 are the same as for sts1. It can hence be concluded that all of the observable behaviours of sts2 are observable behaviours of sts1. It therefore follows that Rule 3 is sound.

The soundness of the second rule could be argued in a similar manner to third rule.

APPENDIX FIVE: Ten Rule deNCE Branching-Looping MASCOT Grammar

A deNCE graph grammar is presented informally in this appendix. It defines the same subset of Branching/Looping MASCOT designs as the BLM_GG given in the body of the thesis, but contains two fewer production rules. It can be argued that this is the simpler grammar, and would be better as the abstract syntax of MASCOT, however, it does not contain the production rules of the SLM_GG as a subset its rules. It is also more ambiguous than the BLM_GG.

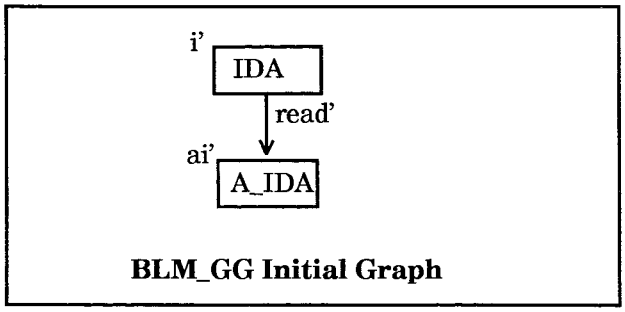



r6
ai'
read*
IDA
read*
::=
p'
write'
pool
read'
r7
s'
write'
signal
read'
r8
c'
write'
channel
read'
r9
write*
IDA
read*
::=
a'
write'
IDA
read'
ACT
i'
a"
write"
IDA
read"
r10
write'
write"
i'
i"
IDA
IDA
read'
read"

APPENDIX SIX: Branching-Looping MASCOT Graph Grammar

A deNCE graph grammar is presented in this appendix for Branching/Looping MASCOT designs, and is called BLM_GG. The Simple Linear MASCOT designs are a subset of such designs, and indeed the production rules of the SLM_GG graph grammar are also part of the BLM_GG grammar. These rules retain their original numbering.

BLM_GG = (Σ, Δ, Z, R), where Σ, the set of all labels, is {ACT, A_IDA, IDA, pool, channel, signal, activity}, Δ, the set of all terminal symbols, is {pool, channel, signal, activity}, and Z, the initial node/edge labelled graph, is ({i' ai'}, {(i', ai')}, {A_IDA, IDA}, {read'}, { (i', IDA), (ai', A_IDA) }, { ((i', ai'), read') }). Z can be presented pictorially as shown below. It is the same initial graph as used in the SLM graph grammar.



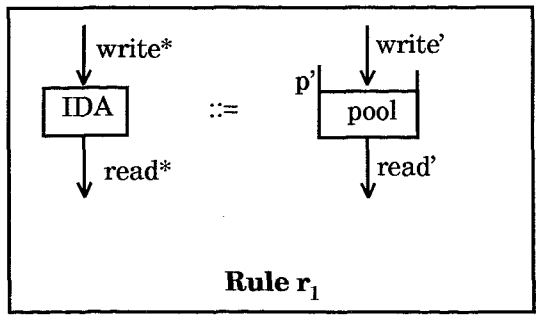
BLM_GG Initial Graph

R, the set of re-writing rules, is { r_i | i $\in$ 1 .. 12 }, where each rule, r_i, takes the form: (x ::= Y, C_{in}, C_{out}), where x $\in \Sigma - \Delta$, Y is an node/edge labelled graph, and $C \subseteq V_Y \times L \times \Sigma \times L$. The rules are repeated below with the daughter graphs presented pictorially. r_1 to r_4 are exactly as for the SLM grammar, except that the embedding relations are extended to handle the new non terminal, ACT. r_5 is modified very slightly, replacing the terminal activity node in Y with a non-terminal ACT node, which can be re-written into an activity using r_9.

The Production Rules

r_1 = (IDA ::= ({p'}, $\emptyset$, {pool}, $\emptyset$, { (p', pool) }, $\emptyset$),
{ (p', write', activity, write*), (p', write', ACT, write*) },
{ (p', read', activity, read*), (p', read', A_IDA, read*),
(p', read', ACT, read*) }).

Pictorially, r_1 is:

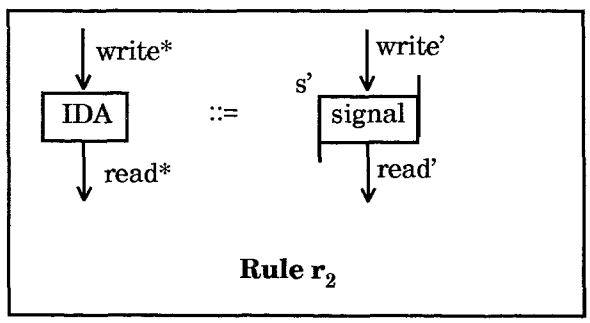


Rule r_1

As before, it should be noted that this pictorial representation does not carry as much embedding information as the textual version of the rule.

r_2 = (IDA ::= ({s'}, $\emptyset$, {signal}, $\emptyset$, { (s', signal) }, $\emptyset$),
{ (s', write', activity, write*), (s', write', ACT, write*) },
{ (s', read', activity, read*), (s', read', A_IDA, read*),
(s', read', ACT, read*) }).

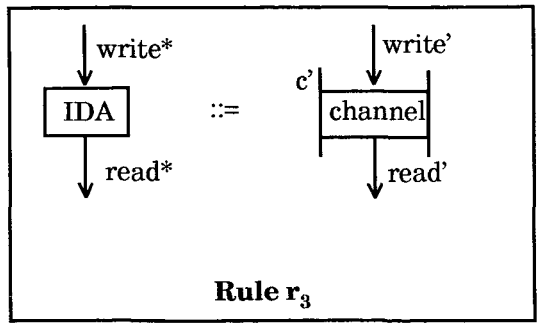
Pictorially, r_2 is:



Rule r_2

r_3 = (IDA ::= ({c'}, $\emptyset$, {channel}, $\emptyset$, { (c', channel) }, $\emptyset$),
{ (c', write', activity, write*), (c', write', ACT, write*) },
{ (c', read', activity, read*), (c', read', ACT, read*),
(c', read', A_IDA, read*) }).

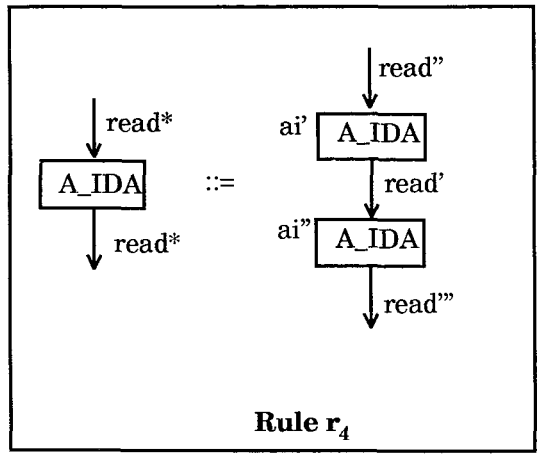
Pictorially, r_3 is:



Rule r_3

r_4 = (A_IDA ::= ({ai', ai"}, { (ai', ai") }, {A_IDA}, {read'},
{ (ai', A_IDA), (ai", A_IDA) }, { ((ai', ai"), read') }),
{ (ai', read", channel, read*), (ai', read", pool, read*),
(ai', read", signal, read*), (ai', read", IDA, read*),
(ai', read", A_IDA, read*) },
{ (ai", read"', activity, read*), (ai", read"', A_IDA, read*),
(ai", read"', ACT, read*) }).

Pictorially, r_4 is:

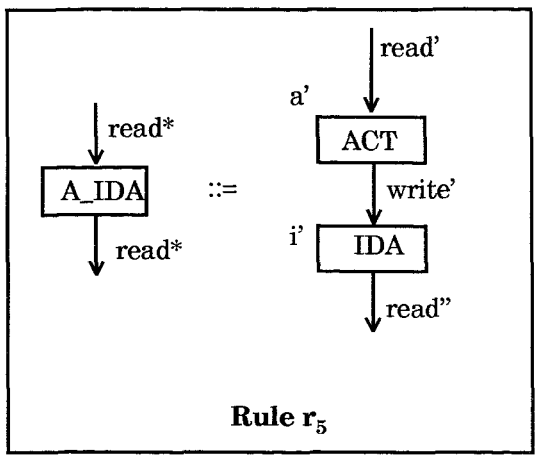


Rule r_4

r_5 = (A_IDA ::= ({a', i'}, { (a', i') }, {ACT, IDA}, {write'}, { (a', ACT), (i', IDA) },
{ ((a', i'), write') }),
{ (a', read', channel, read*), (a', read', pool, read*), (a', read', signal, read*),
(a', read', IDA, read*), (a', read', A_IDA, read*) },
{ (i', read", activity, read*), (i', read", A_IDA, read*),
(i', read", ACT, read*) }).

The node names a' and i' on the right hand side of r_5 are chosen on each application of the rule to be unique in the host graph.

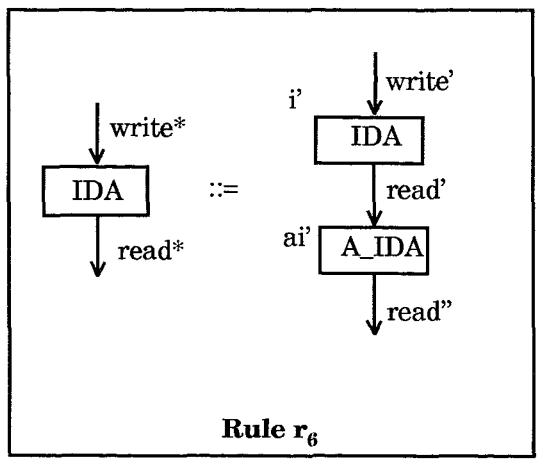
Pictorially, r_5 is:



Rule r_5

r_6 = (IDA ::= ({ai', i'}, { (i', ai') }, {A_IDA, IDA}, {read'}, { (ai', A_IDA), (i', IDA) },
{ ((i', ai'), read') }),
{ (i', write', activity, write*), (i', write', ACT, write*) },
{ (ai', read", activity, read*), (ai', read", A_IDA, read*),
(ai', read", ACT, read*) }).

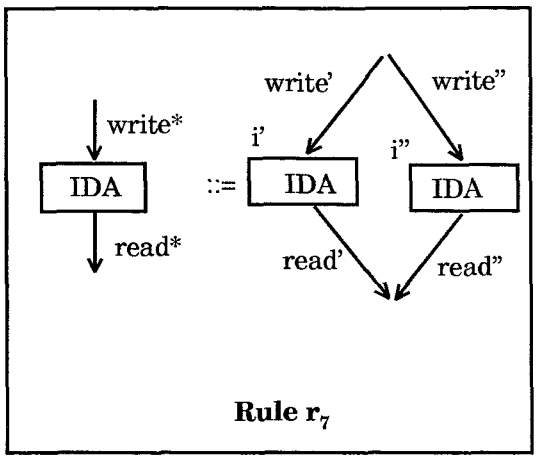
Pictorially, r_6 is:



Rule r_6

r_7 = (IDA ::= ({i', i"}, ∅, {IDA}, ∅, { (i', IDA), (i", IDA) }, ∅),
{ (i', write', activity, write*), (i', write', ACT, write*),
(i", write", activity, write*), (i", write", ACT, write*) },
{ (i', read', activity, read*), (i', read', A_IDA, read*), (i', read', ACT, read*),
(i", read", activity, read*), (i", read", A_IDA, read*),
(i", read", ACT, read*) }).

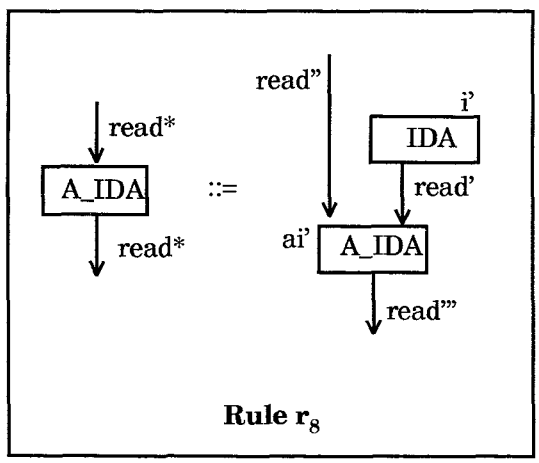
Pictorially, r_7 is:



Rule r_7

r_8 = (A_IDA ::= ({ai', i'}, { (i', ai') }, {A_IDA, IDA}, {read'}, { (ai', A_IDA), (i', IDA) },
{ ((i', ai'), read') }),
{ (ai', read", channel, read*), (ai', read", pool, read*),
(ai', read", signal, read*), (ai', read", IDA, read*),
(ai', read", A_IDA, read*) },
{ (ai', read"', activity, read*), (ai', read"', A_IDA, read*),
(ai', read"', ACT, read*) }).

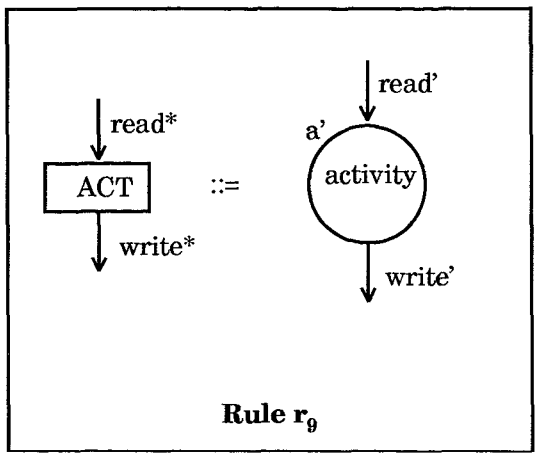
Pictorially, r_8 is:



Rule r_8

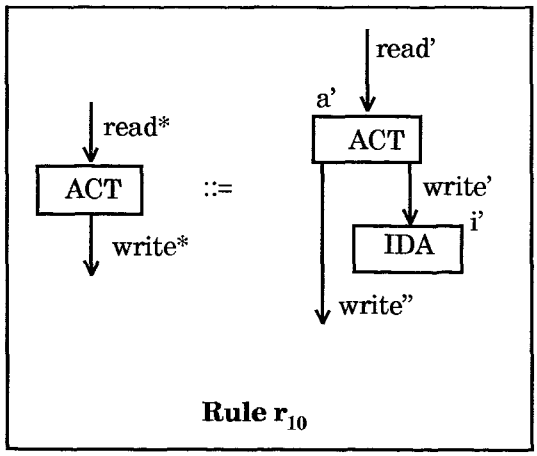
r_9 = (ACT ::= ({a'}, $\emptyset$, {activity}, $\emptyset$, { (a', activity) }, $\emptyset$),
{ (a', read', channel, read*), (a', read', pool, read*), (a', read', signal, read*),
(a', read', IDA, read*), (a', read', A_IDA, read*) },
{ (a', write', channel, write*), (a', write', pool, write*),
(a', write', signal, write*), (a', write', IDA, write*) }).

Pictorially, r_9 is:



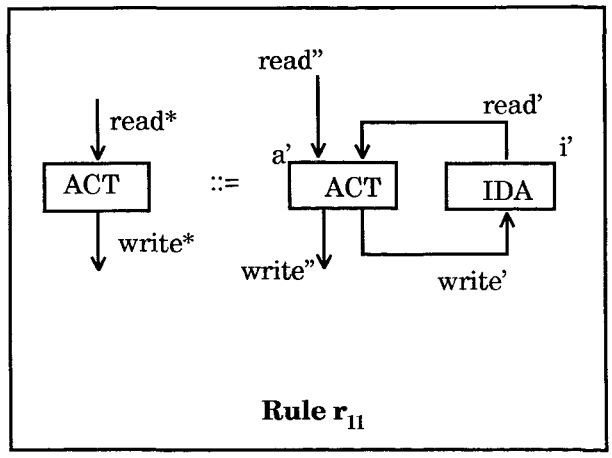
r_{10} = (ACT ::= ({a', i'}, { (a', i') }, {ACT, IDA}, {write'}, { (a', ACT), (i', IDA) },
{ ((a', i'), write') }),
{ (a', read', channel, read*), (a', read', pool, read*), (a', read', signal, read*),
(a', read', IDA, read*), (a', read', A_IDA, read*) },
{ (a', write", channel, write*), (a', write", pool, write*),
(a', write", signal, write*), (a', write", IDA, write*), }).

Pictorially, r_{10} is:



r_{11} = (ACT ::= ({a', i'}, { (i', a'), (a', i') }, {ACT, IDA}, {read', write'},
{ (a', ACT), (i', IDA) },
{ ((i', a'), read'), ((a', i'), write') }),
{ (a', read", channel, read*), (a', read", pool, read*), (a', read", signal, read*),
(a', read", IDA, read*), (a', read", A_IDA, read*) },
{ (a', write", channel, write*), (a', write", pool, write*),
(a', write", signal, write*), (a', write", IDA, write*) }).

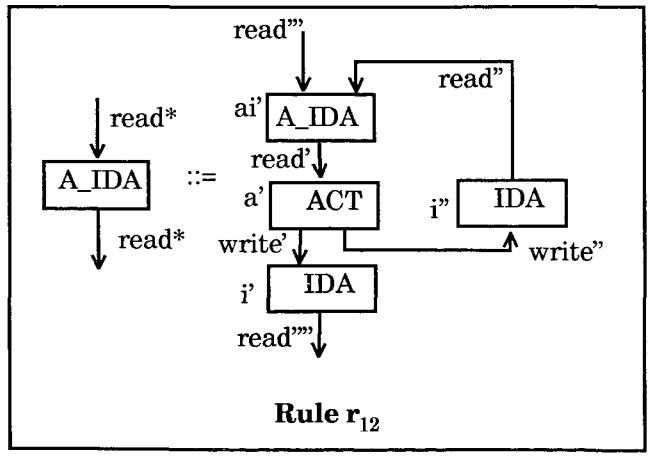
Pictorially, r_{11} is:



Rule r_{11}

r_{12} = (A_IDA ::= ({a', ai', i', i"}, { (ai', a'), (a', i'), (a', i"), (i", ai') },
{ACT, A_IDA, IDA}, {read', read", write', write"},
{ (ai', A_IDA), (a', ACT), (i', IDA), (i", IDA) },
{ ((ai', a'), read'), ((a', i'), write'), ((a', i"), write"), (i", ai'), read") }),
{ (ai', read"', channel, read*), (ai', read"', pool, read*),
(ai', read"', signal, read*), (ai', read"', IDA, read*),
(ai', read"', A_IDA, read*) },
{ (i', read"", A_IDA, read*), (i', read"", ACT, read*),
(i', read"", activity, read*) }).

Pictorially, r_{12} is:



Rule r_{12}

The language of this grammar, L(BLM_GG), is the set of all Branching/Looping MASCOT designs. This class of designs will be given a formal semantics in chapter 5, and of course, in the process, SLM designs will be given a semantics.

REFERENCES

[ABR93] Audsley, N.C., Burns, A., Richardson, M.F., Scholefield, D.J., Wellings, A.J., and Zedan, H.S.M.: "Bridging the Gap Between Formal Methods and Scheduling Theory", Technical Report, YCS 195, Department of Computer Science, University of York, 1993.

[Ace92] Aceto, L.: "Action Refinement in Process Algebras", Distinguished Dissertations in Computer Science, Cambridge University Press, 1992.

[AFK87] Apt, K.R., Francez, N., and Katz, S.: "Appraising Fairness in Languages for Distributed Programming", Proceedings of the 14th ACM[41] Symposium on Principles of Programming Languages, ACM, 1987.

[AlR91] Alagar, V.S. and Ramanathan, G.: "Functional Specification and Proof of Correctness for Time Dependent Behaviour of Reactive Systems", Formal Aspects of Computing, Vol. 3, No. 3, July-September, 1991.

[Arm92] Armstrong, J.: "The Foundations of RTL[42] Expressed as Timing Diagrams", BAe, Dependable Computing Systems Centre, Ref. DCSC/TN/92/7, 1992.

[Ash88] Ashworth, C.M.: "Structured Systems Analysis and Design Method (SSADM)", Information and Software Technology, Vol. 30, No. 3, pages 153-163, April 1988.

[Avn90] Avnur, A.: "Finite State Machines for Real-Time Software Engineering", Computing and Control Journal, November 1990.

[BaM92] Barroca, L.M., and McDermid, J.A.: "Formal Methods: Use and Relevance for the Development of Safety-Critical Systems", The Computer Journal, Vol. 35, No. 6, December 1992.

[Bat86] Bate, G.: "MASCOT-3: An Informal Introductory Tutorial", Software Engineering Journal, Vol. 1, No. 3, May 1986.

[41] Association of Computing Machinery

[42] Real-Time Logic

[BaW90a] Baeten, J.C.M. and Weijland, W.P.: "Process Algebra", Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[BaW90b] Barr, M. and Wells, C.: "Category Theory for Computing Science", Prentice Hall International Series in Computer Science, 1990.

[BCM89] Backhouse, R., Chisholm, P., Malcolm, G., and Saaman, E.: "Do-it-Yourself Type Theory", Formal Aspects of Computing, Vol. 1, No. 1, pages 19-84, January - March 1989.

[Ben92] Benjamin, M.: "The Formal Verification of a Four-Slot Asynchronous Communication Mechanism", Sowerby Research Centre, British Aerospace. February 1992.

[Ber81] Bergland, G.D.: "A Guided Tour of Program Design Methodologies", Computer, IEEE[43], October 1981.

[BJP87] Broy, M., Jackson, K., and Pennington, R.: "A Stream Function Definition of MASCOT", Final Technical Report, Issue 1.1, Systems Designers, Software Technology Centre, March 1987.

[BKT84] Bergstra, J.A., Klop, J.W., and Tucker, J.V.: "Process Algebra with Asynchronous Mechanisms", In "Seminar on Concurrency", Lecture Notes in Computer Science, Vol. 197, Springer-Verlag, July 1984.

[BMP89] Bauer, F.L., Möller, B., Partsch, H., and Pepper, P.: "Formal Program Construction by Transformations - Computer-Aided, Intuition-Guided Programming", IEEE Transactions on Software Engineering, Vol. 15, No. 2, pages 165-180, February 1989.

[Boo86] Booch, G.: "Object-Oriented Development", IEEE Transactions on Software Engineering, Vol. 12, No. 2, pages 211-221, February 1986.

[Boo91] Booch, G.: "Object-Oriented Design with Applications", Benjamin / Cummings Publishing Company, 1991.

[43] Institute of Electrical and Electronic Engineers

[Bro83] Broy, M.: "Applicative Real-Time Programming", In R.E.A. Mason (editor), Information Processing 83, pages 259-264, 1983.

[Bro91] Broy, M.: "Formal Treatment of Concurrency and Time", In "Software Engineer's Reference Book", Edited by J.A. McDermid, Butterworth-Heinemann, 1991.

[BrJ92] Brock, N.A.: "Formal Verification of Fault-Tolerant Computer", IEEE, Proceedings 0-7803-0820-4/92, 1992.

[BrR84] Brookes, S.D. and Roscoe, A.W.: "An Improved Failures Model for Communicating Processes", In "Seminar on Concurrency", Lecture Notes in Computer Science, Vol. 197, Springer-Verlag, 1984.

[BSY89] Bevier, W.R., Smith, M.K., and Young, W.D.: Letters in Reply to "Program Verification the Very Idea", by J. Fetzer, ACM Forum, Communications of the ACM, Vol. 32, No. 3, March 1989.

[Cam86] Cameron, J.R.: "An Overview of JSD", IEEE Transactions on Software Engineering, Vol. 12, No. 2, pages 222-240, February 1986.

[Car79] Carré, B.A.: "Graphs and Networks", Oxford Applied Mathematics and Computing Science Series, Clarendon Press, 1979.

[Car90] Carré, B.A.: "Reliable Programming in Standard Languages", In "High Integrity Software", Edited by C. Sennett, Pitman Publishing, 1990.

[CER79] Claus, V., Ehrig, H., and Rozenberg, G. (Editors): "Graph Grammars and Their Application to Computer Science and Biology", Proceedings of the 1st International Workshop, Lecture Notes in Computer Science, Vol. 73, Springer-Verlag, 1979.

[CGW91] Cullyer, W.J., Goodenough, S.J., and Wichmann, B.A.: "The Choice of Computer Languages for Use in Safety-Critical Systems", Software Engineering Journal, Vol. 6, No. 2, March 1991.

[Coo92] Cooke, J.: "Formal methods - Mathematics, Theory, Recipes or what?", The Computer Journal, Vol. 35, No. 5, pages 419-423, October 1992.

[DaS89] Davies, J. and Schneider, S.: "An Introduction to Timed CSP", Technical Report PRG-75, Oxford University Computing Laboratory, Programming Research Group, August 1989.

[Dav91] Davies, J.: "Specification and Proof in Real-Time Systems", DPhil Thesis, Technical Monograph PRG-93, Oxford University Computing Laboratory, Programming Research Group, April 1991.

[Dij75] Dijkstra, E.W.: "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", Communications of the ACM, Vol. 18, No. 8, pages 453-457, August 1975.

[DLP79] DeMillo, R.A., Lipton, R.J., and Perlis, A.J.: "Social Processes and Proofs of Theorems and Programs", Communications of the ACM, Vol. 22, No. 5, pages 271-280, May 1979.

[EKR90] Ehrig, H., Kreowski, H.,-J., and Rozenberg, G. (Editors): "Graph Grammars and Their Application to Computer Science", Proceedings of the 4th International Workshop, Lecture Notes in Computer Science, Vol. 532, Springer-Verlag, 1990.

[ENR83] Ehrig, H., Nagl, M., and Rozenberg, G. (Editors): "Graph Grammars and Their Application to Computer Science", Proceedings of the 2nd International Workshop, Lecture Notes in Computer Science, Vol. 153, Springer-Verlag, 1983.

[ENR86] Ehrig, H., Nagl, M., Rozenberg, G., and Rosenfeld, A. (Editors): "Graph Grammars and Their Application to Computer Science", Proceedings of the 3rd International Workshop, Lecture Notes in Computer Science, Vol. 291, Springer-Verlag, 1986.

[EnR90] Engelfriet, J., and Rozenberg, G.: "Graph Grammars based on Node Rewriting: An Introduction to NLC Graph Grammars", In [EKR90], 1990.

[Fei93] Feijs, L.: "A Formalisation of Design Methods: A λ-calculus Approach to System Design with an Application to Text Editing", Ellis Horwood Series in Computers and their Applications, 1993.

[Fet88] Fetzer, J.H.: "Program Verification: The Very Idea", Communications of the ACM, Vol. 31, No. 9, pages 1048-1063, September 1988.

[FS93] Formal Systems (Europe) Ltd.: "Announcing Failures Divergences Refinement (FDR): A Verification Tool for Finite State Systems", 3 Alfred Street, Oxford, OX1 4EH, 1993.

[Fuc92] Fuchs, N.E.: "Specifications are (preferably) Executable", Software Engineering Journal, Vol. 7, No. 5, pages 323-334, September 1992.

[Gen87] Genrich, H.J.: "Predicate / Transition Nets", In "Petri-Nets: Central Models and Their Properties", Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, 1987.

[Göd31] Gödel, K.: "On Formally Undecidable Propositions of Principia Mathematica and Related Systems I", Translated by Jean van Heijenoort, Reprinted in "Gödel,'s Theorem in Focus", Edited by S.G. Shanker, Croom Helm Philosophers in Focus Series, Croom Helm, 1988.

[GoF91] Goldsack, S.J. and Finkelstein, A.C.W.: "Requirements Engineering for Real-Time Systems", Software Engineering Journal, Vol. 6, No. 3, May 1991.

[GoT79] Goguen, J.A. and Tardo, J.J.: "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications", Proceedings of Reliable Software, IEEE Catalog No. 79, 1979.

[Haa78] Haack, S.: "Philosophy of Logics", Cambridge University Press, 1978.

[HaJ89] Hayes, I.J. and Jones, C.B.: "Specifications are not (necessarily) Executable", Software Engineering Journal, Vol. 4, No. 5, pages 330-338, November 1989.

[Har87] Harel, D.: "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, No. 8, 1987.

[Har88] Harel, D.: "On Visual Formalisms", Communications of the ACM, Vol. 31, No. 5, May 1988.

[Har91] Harding, M.D.: "User Guide for the MASCOT-3 Design Generator (MADGE)", Digital Information Processing Systems and Technology Department, British Aerospace (Dynamics) Limited, Bristol, BT24610, Issue 4, October 1991.

[He89] He, J.: "Process Simulation and Refinement", Formal Aspects of Computing, Vol. 1, No. 3, July - September 1989.

[Hen88] Hennessy, M.: "Algebraic Theory of Processes", MIT Press Series in the Foundations of Computing, MIT Press, 1988.

[Hen89] Henson, M.C.: "Program Development in the Constructive Set Theory TK", Formal Aspects of Computing, Vol. 1, No. 2, pages 173-192, April - June 1989.

[Hen90] Hennessy, M.: "The Semantics of Programming Languages: An Elementary Introduction using Structured Operational Semantics", John Wiley and Sons, 1990.

[HHS86] He, J., Hoare, C.A.R., and Sanders, J.W.: "Data Refinement Refined - Resume", In "ESOP'86[44]", Lecture Notes in Computer Science, Vol. 213, Springer-Verlag, 1986.

[Hoa69] Hoare, C.A.R.: "An Axiomatic Basis for Computer Programming", Communications of the ACM, Vol. 12, No. 10, pages 576-580, and 583, October 1969.

[Hoa78] Hoare, C.A.R.: "Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, August 1978.

[Hoa85] Hoare, C.A.R.: "Communicating Sequential Processes", Prentice Hall International Series in Computer Science, 1985.

[HodR86] Hooman, J. and de Roever, W.-P.: "The Quest Goes On: A Survey for Proof Systems for Partial Correctness of CSP", In "Current Trends in Concurrency: Overviews and Tutorials", Lecture Notes in Computer Science, Vol. 224, Springer-Verlag, 1986.

[HOH91a] Hull, M.E.C., O'Donoghue, P.G., and Hagan, B.J.: "Development Methods for Real-Time Systems", The Computer Journal, Vol. 34, No. 2, April 1991.

[44] European Symposium on Programming

[HOH91b] Hull, M.E.C., O'Donoghue, P.G., and Hagan, B.J.: "MOON - Modular Object-Oriented Notation", Software Engineering Journal, Vol. 6, No. 1, January 1991.

[Hoo91] Hooman, J.: "Specification and Compositional Verification of Real-Time Systems", Lecture Notes in Computer Science, Vol. 558, Springer-Verlag, 1991.

[HoU79] Hopcroft, J.E. and Ullman, J.D.: "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley Publishing Company, 1979.

[HoW92] ter Hofstede, A.H.M. and van der Weide, T.P.: "Formalisation of Techniques: Chopping Down the Methodology Jungle", Information and Software Technology, Vol. 34, No. 1, January 1992.

[HuC72] Hughes, G.E. and Cresswell, M.J.: "An Introduction to Modal Logic", University Paperbacks, Routledge, 1972.

[IED91] "SPIRITS: Supporting Predictable Implementation of Requirements on Timing and Safety", IED4/1/2133, IEATP Second Call, Full Proposal, BAe (Dynamics) Ltd., SD Europe Ltd., and University of York, 1991.

[INM88] INMOS Ltd.: "Occam 2 Reference Manual", Prentice Hall International Series in Computer Science, 1988.

[IOW90] Ibrahim, R.L., Ogden, J.A., and Williams, S.A.: "Should Concurrency be Specified?", Specification and Verification of Concurrent Systems, Edited by C. Rattray, Workshops in Computing, Springer-Verlag, 1990.

[Jac90] Jackson, K.: "Design Methods for Real-Time Software", IEE[45] Colloquium on "MASCOT and Related Issues", December 1990.

[Jen87] Jensen, K.: "Coloured Petri-Nets", In "Petri-Nets: Central Models and Their Properties", Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, 1987.

[JHH89] Josephs, M.B., Hoare, C.A.R., and He, J.: "A Theory of Asynchronous Processes", Technical Report PRG-TR-6-89, Oxford University Computing Laboratory, Programming Research Group, February 1989.

[45] Institution of Electrical Engineers

[JIM83] Joint IECCA[46] and MUF[47] Committee on MASCOT (JIMCOM): "The Official Handbook of MASCOT", MASCOT II, Issue 2, Crown Copyright, June 1983.

[JIM87] Joint IECCA and MUF Committee on MASCOT (JIMCOM): "The Official Handbook of MASCOT", Version 3.1, Issue 1, Crown Copyright, June 1987.

[JLH91] Jaffe, M.S., Leveson, N., Heimdahl, M.P.E., and Melhart, B.E.: "Software Requirements Analysis for Real-Time Software Control", IEEE Transactions on Software Engineering, Vol. 17, No. 3, March 1991.

[Jon83] Jones, C.B.: "Specification and Design of (Parallel) Programs", In Proceedings of IFP'83, pages 321-332, North-Holland, 1983.

[Jon90] Jones, C.B.: "Systematic Software Development Using VDM: 2nd Edition", Prentice-Hall International Series in Computer Science, 1990.

[Jos88] Josephs, M.B.: "A State-Based Approach to Communicating Processes", Distributed Computing, 39-18, 1988.

[Kap90] Kaplan, S.M.: "Applying Graph Grammars to Software Engineering", Part of "Panel Discussion: The Use of Graph Grammars in Applications", In [EKR90], 1990.

[KiP85] King, M.J. and Pardoe, J.P.: "Program Design Using JSP: A Practical Introduction", Macmillan Computer Science Series, Macmillan Education Ltd., 1985.

[Kno90] Knowles, R.: "Mapping a MASCOT-3 Design into Occam", Software Engineering Journal, Vol. 5, No. 4, July 1990.

[Koy89] Koymans, R.L.C.: "Specifying Message Passing and Time-Critical Systems with Temporal Logic", Ph.D. Thesis, Technische Universiteit Eindhoven, 1989.

[Kwi89] Kwiatowska, M.Z.: "Event Fairness and Non-Interleaving Concurrency", Formal Aspects of Computing, Vol. 1, No. 3, July-September 1989.

[46] Inter Establishment Committee on Computer Applications

[47] MASCOT Users' Forum

[Kwi91] Kwiatowska, M.Z.: "On Topological Characterisation of Behavioural Properties", In "Topology and Category Theory in Computer Science", Oxford Science Publications, Clarendon Press, 1991.

[Lak76] Lakatos, I.: "Proofs and Refutations: The Logic of Mathematical Discovery", Edited by J. Worrall and E. Zahar, Cambridge University Press, 1976.

[Led90] Ledru, Y.: "Hierarchical Specification of Reactive Systems: A Case Study", CH2867-0, IEEE, pages 109-116, 1990.

[LRM83] "The Ada Language Reference Manual", ANSI[48] Standard, January 1983.

[MaP81] Manna, Z. and Pnueli, A.: "Verification of Concurrent Programs: The Temporal Framework", pages 215-273, In "The Correctness Problem in Computer Science", Edited by R.S. Boyer and J.S. Moore, International Lecture Series in Computer Science, Academic Press, 1981.

[MaP92] Manna, Z. and Pnueli, A.: "The Temporal Logic of Reactive and Concurrent Systems: Specification", Springer-Verlag, 1992.

[Mar82] Martin-Löf, P.: "Constructive Mathematics and Computer Programming", In "Proceedings of the 6th International Congress for Logic, Methodology and Philosophy of Science", pages 153-175, Studies in Logic and the Foundations of Mathematics Vol. 104, North-Holland, 1982.

[Mid89a] Middelburg, M.A.: "Syntax and Semantics of VVSL: A Language for Structured VDM Specifications", PhD Thesis, Univeriteit van Amsterdam, 1989.

[Mid89b] Middelburg, M.A.: "VVSL: A Language for Structured VDM Specifications", Formal Aspects of Computing, Vol. 1, No. 1, January-March 1989.

[Mid92] Middelburg, M.A.: "Specification of Interfering Programs Based on Interconditions", Software Engineering Journal, Vol. 7, No. 3, May 1992.

[Mil89] Milner, R.: "Communication and Concurrency", Prentice Hall International Series in Computer Science, 1989.

[48] American National Standards Institute

[MoD85] Ministry of Defence, "Modular Approach to Software Operation and Test - MASCOT", Defence Standard 00-17, Issue 1, October 1985.

[MoD91a] Ministry of Defence, Sea Systems Controllerate: "Requirements for Software for use with Digital Processors, Naval Engineering Standard, NES 620, Issue 4, June 1991.

[MoD91b] Ministry of Defence: "The Procurement of Safety-Critical Software in Defence Equipment", Interim Defence Standard 00-55, Technical Report, 1991.

[Moo90] Moore, A.P.: "The Specification and Verified Decomposition of System Requirements Using CSP", IEEE Transactions on Software Engineering, Vol. 16, No. 9, September 1990.

[Mor90] Morgan, C.C.: "Programming from Specifications", Prentice Hall International Series in Computer Science, 1990.

[MRG88] Morgan, C.C., Robinson, K.A., and Gardiner, P.H.B.: "On the Refinement Calculus", Technical Monograph, PRG-70, Oxford University Computing Laboratory Programming Research Group, October 1988.

[MuP92] Murphy, D., and Pitt, D.: "Real-Timed Concurrent Refinable Behaviours", In "Proceedings of the Second International Symposium of Formal Techniques in Real-Time and Fault-Tolerant Systems", Lecture Notes in Computer Science, Vol. 571, Springer-Verlag, 1992.

[Mur91a] Murphy, D.: "Observers, Philosophers, and Spacetime", In "3 Papers on Classical Concurrency Theory", Technical Report 91/R5, University of Glasgow, May 1991.

[Mur91b] Murphy, D.: "The Physics of Observation: A Perspective for Concurrency Theorists", Bulletin of the EATCS[49], No. 44, June 1991.

[MVL92] Misic, V., Velasevic, D., and Lazarevic, B.: "Formal Specification of a Data Dictionary for an Extended ER[50] Data Model", The Computer Journal, Vol. 35, No. 6, December 1992.

[49] European Association of Theoretical Computer Science

[50] Entity-Relationship

[Nie92] Nielsen, M.: "Models for Concurrency", Tutorial Notes, Presented at Petri-Nets'92, Sheffield, England, July 1992.

[NiN92] Nielson, H.R, and Nielson, F.: "Semantics With Applications: A Formal Introduction", Wiley Professional Computing, John Wiley and Sons, 1992.

[Pag81] Pagan, F.G.: "Formal Specification of Programming Languages: A Panoramic Primer", Prentice-Hall, 1981.

[Par90] Partsch, H.A.: "Specification and Transformation of Programs: A Formal Approach to Software Development", Texts and Monographs in Computer Science, Springer-Verlag, 1990.

[Pnu86a] Pnueli, A.: "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends", In "Current Trends in Concurrency: Overviews and Tutorials", Lecture Notes in Computer Science, Vol. 224, Springer-Verlag, 1986.

[Pnu86b] Pnueli, A.: "Specification and Development of Reactive Systems", In "Information Processing 86", Elsevier Science Publishers, 1986.

[Rei87] Reisig, W.: "Place / Transition Systems", In "Petri-Nets: Central Models and Their Properties", Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, 1987.

[Rob88] Robinson, P.J.: "HOOD Manual - Issue 2.2", European Space Agency, ESTEC, April 1988.

[Roz86] Rozenberg, G.: "An Introduction to the NLC Way of Rewriting Graphs", In [ENR86], 1986.

[Sch86] Schmidt, D.A.: "Denotational Semantics: A Methodology for Language Development", Wm. C. Brown Publishers, 1986.

[Sch90] Schneider, S.: "Correctness and Communication in Real-Time Systems", DPhil Thesis, Technical Monograph PRG-84, Oxford University Computing Laboratory Programming Research Group, March 1990.

[SFD92] Semmens, L.T., France, R.B., and Docker, T.W.G.: "Integrated Structured Analysis and Formal Specification Techniques", The Computer Journal, Vol. 35, No. 6, December 1992.

[SiJ79] Simpson, H.R. and Jackson, K.: "Process Synchronisation in MASCOT", The Computer Journal, Vol. 22, No. 4, August 1979.

[Sim86] Simpson, H.R.: "The MASCOT Method", Software Engineering Journal, Vol. 1, No. 3, May 1986.

[Sim90a] Simpson, H.R.: "A Data Interaction Architecture (DIA) for Real-Time Embedded Multi-processor Systems", Royal Aeronautical Society Conference on Computing Techniques in Guided Flight, Boscombe Down, 1990.

[Sim90b] Simpson, H.R.: "Four-Slot Fully Asynchronous Communication Mechanism", IEE Proceedings, Vol. 137, Part E, No. 1, January 1990.

[Sim91] Simpson, H.R.: "Real-Time Networks for Embedded Computing Systems: The DORIS / DIA Approach", British Aerospace (Defence) Ltd., 1991.

[Sim92] Simpson, H.R.: "Correctness Analysis for Class of Asynchronous Communication Mechanisms", IEE Proceedings, Vol. 139, Part E, No. 1, January 1992.

[Sim93] Simpson, H.R.: "Methodological and Notational Conventions in DORIS Real-Time Networks", Draft, February 1993.

[SMC74] Stevens, W.P., Myers, G.J., and Constantine, L.L.: "Structured Design", IBM Systems Journal, No. 2, 1974.

[SrH85] Sridhar, K.T., and Hoare, C.A.R.: "JSD Expressed in CSP", Technical Monograph, PRG-51, Oxford University Computing Laboratory Programming Research Group, July 1985.

[Stø92] Stølen, K.: "Proving Total Correctness with respect to a Fair (Shared State) Parallel Language", In "Proceedings of the Fifth Refinement Workshop", Workshops in Computing, Springer-Verlag, 1992.

[SWB87] Sommerville, I., Welland, R., and Beer, S.: "Describing Software Design Methodologies", The Computer Journal, Vol. 30, No. 2, March 1987.

[Thi87] Thiagarajan, P.S.: "Elementary Net Systems", In "Petri-Nets: Central Models and Their Properties", Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, 1987.

[Tho91] Thomas, C.M.: "The Data Oriented Requirements Implementation Scheme", In "Software for Guidance and Control", AGARD[51] Conference Proceedings 503, North Atlantic Treaty Organisation, 1991.

[TKP90] Toetenel, H., van Katwijk, J., and Plat, N.: "Structured Analysis - Formal Design, using Stream and Object Oriented Formal Specification", Proceedings of the ACM SIGSOFT[52] International Workshop on "Formal Methods in Software Development", May 1990.

[Tse91] Tse, T.H.: "A Unifying Framework for Structured Analysis and Design Methods", Cambridge Tracts in Theoretical Computer Science, No. 11, Cambridge University Press, 1991.

[TsP89] Tse, T.H., and Pong, L.: "Towards a Formal Foundation for DeMarco Data Flow Diagrams", The Computer Journal, Vol. 32, No. 1, February 1989.

[WBS90] Welland, R., Beer, S., and Sommerville, I.: "Method Rule Checking in a Generic Design Editing System", Software Engineering Journal, Vol. 5, No. 2, March 1990.

[Win85] Winskel, G.: "Categories of Models for Concurrency", Seminar on Concurrency, LNCS 197, Springer-Verlag, 1985.

[Win87] Winskel, G.: "Event Structures", Petri Nets: Applications and Relationships to Other Models of Concurrency. LNCS 255, Springer-Verlag, 1987.

[WiN92] Winskel, G. and Nielsen, M.: "Models for Concurrency", Technical Report, Computer Science Department, Aarhus University, DAIMI PB - 429, November 1992.

[51] Advisory Group for Aerospace Research and Development

[52] Special Interest Group in Software Engineering

[WoM90] Woodcock, J.C.P., and Morgan, C.C.: "Refinement of State-Based Concurrent Systems", In "VDM'90: VDM and Z", Lecture Notes in Computer Science, Vol. 428, Springer-Verlag, 1990.

[Woo88] Woodman, M.: "Yourdon Dataflow Diagrams: A Tool for Disciplined Requirements Analysis", Information and Software Technology, Vol. 30, No. 9, pages 515-533, November 1988.

[XuH91] Xu, Q. and He, J.: "A Theory of State-Based Parallel Programming: Part 1", 4th Refinement Workshop, Edited by J.M. Morris and R.C. Shaw, Workshops in Computing, Springer-Verlag, 1991.

[XuH92] Xu, Q. and He, J.: "A Case Study in Formally Developing State Based Parallel Programs: The Dutch National Torus", 5th Refinement Workshop, January 1992. To be published in the Workshops in Computing series, Springer-Verlag.