ON THE FIDELITY OF SOFTWARE

by

Stephen J. Counsell

A THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in the

UNIVERSITY OF LONDON

February 2002

Birkbeck College

Abstract

A fundamental aspect of software quality and reliability is ensuring that an implementation meets the requirements of a specification. In this thesis, the concept of *software fidelity* is examined. We define fidelity as the closeness of an implementation to its specification at any point during the refinement of that implementation. A set of refinement metrics were introduced to capture features of the implementation as it was refined and an empirical study using four CSP systems was carried out. The four systems covered two application domains. The first two systems were bit-protocol type problems (a multiplexed-buffer problem and the alternating bit protocol problem). The second two systems were classical computer science problems (the Towers of Hanoi and the Dining Philosophers problems). A key result of the analysis showed that the implementations of the two pairs of systems exhibited similar characteristics as they were refined. Significant differences, however, were found when comparing the two application domains, suggesting that different types of problem give rise to different features (given by the set of metrics). The initial set of metrics proposed were only found to be appropriate for bit-protocol problems; the need for flexibility in metrics development to cater for different application domains is another important issue raised by the research. The notation of Communicating Sequential Processes (CSP) and its underlying semantics was used as the vehicle for expressing both specification and implementation(s).

Acknowledgements

I would like to thank a number of people for their support during the time spent on the thesis. Firstly, to my parents for never asking anything from me except that I try my best. To Rob, my elder brother for financial help in the early days; John, my younger brother for help in other ways. To my friends throughout this period, in particular Pete, Alex, Michele Tim, Jackie, Swifty, Allan and Hui. To Stella for putting up with me in very bad times (and good of course). I would also like to thank Jason Crampton and Roger Mitton in the School for reading drafts of the various thesis chapters. I also express my thanks to the Systems Group in the School for their technical support. Lastly, but by no means least, I express my sincere gratitude to George (Professor Loizou), my supervisor at Birkbeck, who I was fortunate to have around and from whom I have learnt a tremendous amount, even when I thought I knew it all.

Contents

List of Figures

Chapter 1

Introduction

One of the major problems in the field of software engineering today is to ensure that a piece of software is engineered to meet its specified requirements. In this thesis, we assume those requirements to be the true requirements of a piece of software [1], although in Chapter 8, we will briefly look at the effect that changing the specification has on its relationship with the implementation. The dominant theme of this thesis is the measurement of differences between a piece of software and its specification. Such measurements establish the degree of *faithfulness* or *fidelity* of a piece of software.

Efficient and affordable improvements in software fidelity have been achieved through the use of:

- specification languages (see [Jon86] and [Spi88]) which allow the capture of requirements using a precise notation;
- various automated model checkers ([BS94] and [For97]) and tools which generate code automatically — so called program generators;

[1] The area of determining whether user requirements have been satisfactorily met is an area currently the subject of much research, and beyond the scope of this current work; the interested reader should consult [Som98].

- languages with specific features geared towards the generation of reliable code; see, for example [Dep81], [US87] and [Bar93][2];
- more recently developed languages such as C++ [Str94], which, through *encapsulation* and *inheritance*, allow data types to be constructed which mirror the natural structure of the application domain to be realised.

All these languages and tools have reduced the gap between specification and implementation, thus making the task of development and maintenance of software quicker and less error-prone. However, we are still faced with the problem that, at the most detailed level, there is unlikely to be an accurate mapping between specification and its implementation.

Henceforth, we will use the term *specification* to describe a set of user requirements, and *implementation* to describe the code generated either automatically, in the case of a program generator, or manually, in the case of a specification language such as Z. We view the problem which this thesis addresses as one of characterising the differences, via a set of metrics, between a specification and its implementation. In the next section, the problem which this thesis addresses is explained.

1.1 The problem explained

In the software engineering world, the problem of ensuring a piece of software satisfies the user requirements of its initial specification is still a very active research area. Use of specification languages to produce reliable real-world systems has been successfully demonstrated [WB95]. However, generally speaking, we are still faced with the problem of characterising and quantifying the differences between a specification and its implementation.

[2]In Chapter 2, we make a distinction between the fidelity of software and the reliability of software; for now, we take the two to mean the same.

Even with the assistance of program proof techniques such as those developed by Hoare [Hoa69] and Owicki and Gries [OG76] and exemplified by the automated program prover of King [Kin77], we can never hope to prove very large systems correct, simply because of their complexity, and the large number of states which are generated as that complexity increases[3]. Even small systems can generate a large number of states, and this presents any comparison of a specification and its implementation with the problem of balancing the cost of generating and checking those states against the benefits accrued by performing the comparison.

1.1.1 A notion of divergence

If we accept that there are going to be differences between a specification and its implementation in terms of the emergent behaviour of the latter, then we can say that an implementation *diverges* from that specification when it exhibits unspecified behaviour; divergence of an implementation from its specification necessarily reduces the fidelity of that implementation. This is not to say, however, that divergent behaviour is necessarily invalid behaviour as will be shown subsequently in the thesis.

The notion of divergence of an implementation from its specification in the sense just described is not a new one. In a concurrent setting, the term *divergent* [BHR84] is frequently used to describe a state in which a process is exhibiting what we usually call an infinite loop; in this thesis, divergence therefore takes on a different slant.

If we accept that failures are likely to occur in all but software of a trivial nature, then it would seem sensible to approach the problem of characterising the differences between a specification and its implementation by first expressing both in some common medium. To

[3]For now, we take the term *state* to mean the result of executing a program statement.

illustrate what we mean, we turn to a simple example expressed in the language of CSP (Communicating Sequential Processes) [Hoa85].

Example

The example considers the specification of a CSP single-place buffer which is implemented using two processes communicating internally. The set of values to be communicated in this system are `apples`, `oranges` and `pears`. The channels down which those values are to be passed are `left`, `right` and `mid`. The `ack` channel is used to acknowledge receipt of a value. The specification, called `COPY`, is a single place buffer which receives a value on the `left` channel and outputs that value on the `right` channel. The implementation, called `SYSTEM`, consists of two processes `SEND` and `REC` which communicate over `mid` and `ack`. The internal (or hidden) behaviour (given by channels `mid` and `ack`) is denoted by the backward slash symbol. Checking the implementation `SYSTEM` against the specification `COPY` will confirm that the implementation does indeed refine the specification, since externally observed, specification and implementation are identical [Bro83].

```
datatype FRUIT = apples | oranges | pears

channel left,right,mid : FRUIT
channel ack

COPY = left ? x -> right ! x -> COPY

SEND = left ? x -> mid ! x -> ack -> SEND
REC = mid ? x -> right ! x -> ack -> REC

SYSTEM = (SEND [| {| mid, ack |} |] REC) \ {| mid, ack |}

assert COPY [FD= SYSTEM
```

A diagrammatic representation of `SYSTEM` can be seen in Figure 1.1. The added (hidden) behaviour incorporated into `SYSTEM` is that inside the boundary of the outer box.

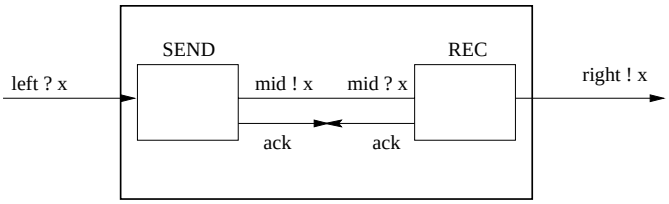


Figure 1.1: Diagram of `SYSTEM`

A number of points arise from looking at this CSP example. Firstly, we could develop the implementation (i.e., `SYSTEM`) by constructing either `SEND` or `REC` first. At any point, we could run a refinement check of `SYSTEM` against `COPY`, i.e., `assert COPY [FD= SYSTEM`, to establish whether refinement [Hoa85] held; if not, the reason for non-refinement could be obtained from the model-checker. Alternatively, we could build the sender and receiver concurrently, adding the hidden behaviour (interface behaviour) at the end. Whichever way was chosen, the proper refinement has appropriate hidden and visible behaviour, so that, externally, both processes appear identical. We could also add further values to the `datatype` `FRUIT`, for example, `plums`, so that we could view the effect on the refinement process this would have. An important point to note here is that the implementation `SYSTEM` can be mapped directly to a language supporting CSP constructs; an example of such a language would be Occam [PM87].

In order to capture information about the behaviour of the implementation (and the level of fidelity with the specification) we need a set of metrics by which the emergent behaviour of the implementation can be monitored and assessed. We also need a way of evaluating the impact of candidate changes to `SYSTEM`, all of which may be feasible, but only one of which may generate a small number of implementation states.

1.2 Motivation

As we have seen, a specification or implementation expressed in CSP can be realised in terms of the low-level communications along channels between a number of processes. In Chapter 3, we define a state and an action, the former arising as a result of the latter. (In the CSP example, the actions were communications along channels). We use the concepts of state and action as the vehicles of our analysis. We then have a foundation for our analysis of divergence and propose a set of metrics for measuring features of both specification and implementation.

Identifying the differences between specification and implementation will give us a measure of the fidelity of differently constructed implementations of the specification. We note that such an approach was first proposed by Counsell [Cou92] and extended in [CM96]. The question that then naturally arises is to why we should be interested in identifying differences between a specification and its implementation during refinement [Bro83]. The main reasons are that an implementation expressed in CSP, as in the example of the previous section, (i.e., conforming to refinement principles) is likely to:

1. contain fewer faults (since the starting point is the specification itself);

2. be a more efficient implementation if it uses refinement in an incremental fashion during the process of development. Each refinement is checked against the specification;

3. is likely to be more maintainable and, since it appeals to software engineering principles [Som98], will require less maintenance.

1.3 Objectives and contribution

The main objectives of the thesis are:

1. To obtain a greater understanding of the behaviour of processes expressed in CSP as they are developed through refinement and non-refinement.

2. To assess the capability of a set of proposed metrics for measuring the differences between a specification and its implementation during refinement, with specific emphasis on the types of state encountered in each.

We now briefly consider to what extent these objectives have been achieved.

Chapter 6 considers two bit-protocol problems, namely, a multiplexed buffer and an alternating bit protocol. We define this type of problem, expressed in CSP, as a Type I CSP system. A significant amount of commonality was found between these two Type I CSP systems; for example, in the propensity for each such system to generate a large number of states. Equally, subtle differences were found between them. For example, the extent to which each such system carried out error condition checking in the body of the CSP code itself.

Chapter 7 looks at two further problems, namely, the Towers of Hanoi problem and the Dining Philosophers problem; we define this type of problem, expressed in CSP, as a Type II CSP system. Significant similarities were also found between these two Type II CSP systems. In particular, the level of recursion generated by each system imposed restrictions on the degree to which the state space could be increased.

With respect to the first objective, the main lesson learnt from the analysis on the four CSP systems was the complete contrast between Type I CSP systems and Type II CSP

systems. The contrast ranged from the style in which each pair of systems was written, the static CSP features used by each such pair and the behaviour of each pair when analysed.

Addressing the second of our objectives, for the bit-protocol oriented problems, the metrics were useful in showing features such as the growth in the state space, the number of sub-processes generated as this occurred and the influence of different types of action on the behaviour of the sub-processes. Interestingly, the set of proposed metrics were not found to be suitable for Type II CSP systems, and thus alternative metrics were suggested in this case.

In terms of greater understanding of the features and behaviour of different types of CSP system, the first objective has been achieved. Although the set of proposed metrics did not show themselves to be applicable across both types of CSP system, this lesson in itself and the opportunity to modify the set of proposed metrics was a valuable experience. Empirical evidence in this thesis suggests that no unified set of metrics exists for different application domains when expressed in CSP. In this sense, the second objective has been realised as well.

1.4 Overview of the thesis

In Chapter 2 we present a survey of related work, indicating the major areas which have influenced our work.

In Chapter 3 we introduce formal definitions of state, action, finite state machine and labelled transition system. These concepts are fundamental in developing the work presented in subsequent chapters.

In Chapter 4 we begin our analysis of divergence with a definition of two types of state. We call these two types of state *non-divergent* and *partially divergent*. We show that the states of a process, on any execution, can be expressed in terms of just these two types

of state, and identify the state transitions (which are possible between these two types of state). We also define process alphabets, information hiding and relabelling. We then consider the notions of a specified action (having a similarity with a non-divergent state) and an unspecified action (having a similarity with a partially divergent state) and formally define these two types of action.

In Chapter 5 we describe the set of metrics which we utilise in estimating the level of divergence, and hence the fidelity, of an implementation given the specification.

In Chapter 6 we analyse the two Type I CSP systems in the context of refinement and non-refinement. These are the Multiplexed Buffer problem and the Alternating Bit Protocol problem. The associated CSP systems for each of these two problems are analysed and compared by using the set of metrics introduced in Chapter 5. This includes an analysis of how each system evolves during refinement in terms of its sub-processes.

In Chapter 7 we analyse the two Type II CSP systems in the context of refinement and non-refinement. These are the Towers of Hanoi problem and the Dining Philosophers problem. We also look briefly at some alternative metrics in the light of the difficulties experienced when trying to apply the set of metrics introduced in Chapter 5. In addition, we also consider the effect, in terms of the metrics values, of introducing mutation into the implementation.

Finally, in Chapter 8, we look briefly at the situation where the specification is modified in order to reflect changes in user requirements and the effect this has on the values of the metrics. We then draw some conclusions and discuss possible future work.

Throughout the thesis the terms *cardinality* and *size* of a set will be used interchangeably. Similarly, the *set of states* or the *size of the state space* (see Chapters 3 and 4) will also be used interchangeably.

Chapter 2

A survey of related work

2.1 Introduction

Before we present our empirical analysis of the four CSP systems and hence attempt to learn more about the emerging behaviour of their implementation through a set of proposed metrics, it is appropriate to describe in some detail some of the work previously carried out in related and complementary areas. The reasons for this are: firstly, it justifies our approach to software fidelity, and, secondly, it demonstrates how we arrived at this approach, taking inspiration from certain areas whilst rejecting others. We examine how our research achieves its objectives in the light of that related work.

In Section 2.2, we discuss the area of formal methods which incorporates the traditional view taken of divergence, specification languages, process equivalence and process refinement, the CCS and CSP models of concurrency and the use of model-checkers. A model-checker was used in this thesis to aid the extraction of metrics and to permit refinement checks upon which the empirical study rests. In Section 2.3, we describe related work in the use and analysis of software metrics, and compare that work with our metrics. Since our work is

empirical, Section 2.3 also describes some relevant work in the empirical software engineering field. For each of these areas, we examine related and complementary work, justifying in each case why our approach is different. Finally, we summarise the contents of the chapter in Section 2.4.

2.2 Formal methods

According to Spivey [Spi88], formal methods comprise two things: *formal specification* and *verified design*. The first denotes the precise specification of the behaviour of a piece of software, the second that of proving that an implementation meets its specification. In our model, we are primarily concerned with the second. This is because we assume the specification satisfies the stated requirements of the problem. Chapter 8 includes some tentative work in which we try to assess the effects on the specification as a result of a change request, but delving in greater depth at this aspect of software engineering is an area for significant future research.

Unlike most other specification languages, CSP allows a concrete (low-level) implementation to be derived from an abstract definition of a specification. A major drawback with many specification languages is that although they permit an abstract view of a specification, this does not lend itself well to the automatic production of a concrete implementation. Choice of a specification language to use therefore has a major impact on the ease with which an implementation can be produced. The research contained herein is underpinned by the assumption that specification and implementation are expressed using the same medium, namely, CSP. This differentiates the work contained in this thesis from previous work [KvEvS90, AM94, BDFM99].

One specification language, which induces a straightforward mapping from a specification

to an implementation, is the Z specification language [Spi88]. Other specification languages are VDM [BJ82], LOTOS [FSV92] and LCS [BS94]. Of particular interest is Z, in which the notions of a *before* and an *after* state of an event are made explicit, as is the concept of state identifier values changing as a result of a state transition (alluded to in the model developed in this thesis).

There has also been some work done in the automatic conversion of Z to an implementation language. Specifically, Spivey [Spi88] describes the *animation* of Z specifications in both Miranda [Tur86] and Prolog [Bra90]. Similarly, Woodcock and Morgan [WM90] describe how Z can be translated into CSP. Some work has also been undertaken in code generation from an object-oriented viewpoint [BFVY96].

Over the past ten years, a body of research has also been built up devoted to extracting metrics from Z specifications [Spi88]. We adopt a similar approach when extracting metrics from CSP systems (see Chapters 6 and 7). We illustrate some of the characteristics of Z via a simple example.

Example

Consider a library book lending service. The state of such a system can be modelled at any moment by the number of books on the shelves plus any books returned, and hence waiting to be re-shelved. These represent the two variables of our system. The return of a single book will cause the number of books waiting to be re-shelved to increase by one. However, the number of shelved books remains unchanged by this event. Similarly, if all books in the returned stacks are placed back on the shelves then both variables of our system change (i.e., books on shelves and books returned).

A Z specification comprises a top-level schema and associated sub-schemas. The latter

can be likened to sub-processes of an overall process. The schema defines the top-level functions which the specification uses, and the sub-schemas define the operations which each of those functions perform[1]. A Z schema actually fulfils two roles: firstly, to describe the possible states of a process and, secondly, to describe the state transitions (which cause those states to change), and the effect on any identifiers used by the specification of that state change.

There is a close relationship between the way Z specifies how an implementation should behave and the concepts of a non-divergent and partially divergent state described in Chapter 4 as part of our model. The emphasis in the Z language and encapsulated in our two types of state (see Chapter 1) is on the nature of the events (or actions as we call them) causing state transitions. A state transition in our model causes changes in the values held in a single identifier. Our model, unlike previous work, captures state information from both the specification and the implementation; this information is then used as a basis for the analysis of the differences between the two.

In [Whi90] the problem of measuring Z specifications is addressed, complementing earlier work in the area of complexity measures [Pra84, FW86]. A *short circuit* model for evaluating the structure of Z specifications is described using a graph-based system to describe Z expressions. The interesting feature of this technique is that it focuses on the important aspects of a typical Z specification (i.e., universal and existential quantifiers). The same is true for the metrics we develop for CSP systems (see Chapter 5), with the emphasis on states, actions and sub-processes – features which we felt were representative of every CSP system. In keeping with this theme of extracting metrics from specifications, an approach in which metrics could be collected at an early stage from specifications (using automated

[1] We could say that the schema defines *what* the functions of the specification are, and the sub-schemas *how* those functions are realised.

tools) was the subject of work undertaken by the COSMOS (Cost Management with Metrics for Specifications) project [FTW90]. The idea was that early identification of problems in the development process through the automated production of graphs and visual aids could help to prevent some of the problems normally associated with software development from occurring, e.g., cost overruns, time delays, etc. In this thesis, the focus of our research is on the identification of features of CSP system behaviour as a means of determining how well the implementation refines the specification at every stage throughout the development of the implementation. Description of a tool to collect metrics directly from Z specifications, description of the metrics themselves, and analysis of the values obtained from some sample specifications, as an enactment of the COSMOS objectives, are given in [BWW91].

In [SDN+89] the relationship between specification and implementation was examined for pairs of specification and implementation. For each specification, an implementation, cast in another language, was analysed for correlation with its specification. Results showed that the lines of code in an implementation could be predicted from certain features of the specification. Additionally, and interestingly, the programming style of the implementation was found to be independent of the program features necessary to do the task required. In other words, rigorous techniques need to be adopted in the implementation as well as in the specification in order for similar implementations to be produced. This is supported by a finding we have discovered via experimentation for both Type I and Type II CSP systems, i.e., that certain CSP constructs would always have to be used in the implementation for certain types of application, whatever style of programming in CSP was used. This is because they are constructs essential for functionality to be expressed. For example, both the bit-protocol systems analysed in Chapter 6 used significant amounts of communication operators; this was not the case for the Towers of Hanoi and Dining Philosophers problems of Chapter

7, because these two problems contain features intuitively and inherently more related to recursion. Our research therefore reinforces the findings in [SDN+89] and reinforces the need for rigour in both the specification and implementation.

Of particular interest in the area of formal methods are the tools becoming available for converting from specification languages to concrete implementation languages. In this thesis, we use the FDR model-checker to produce the underlying graphs of specification and implementation (each CSP system is represented internally by a graph structure). Therefrom, we are able to enumerate the differences between the two. Of related interest is [O'N92], in which it becomes possible to convert from VDM to SML. Similarly, Arrowsmith and McMillin [AM94] describe a system for debugging distributed systems; the system converts from CSP to C. The specification language LOTOS [FSV92] has proved useful in developing and designing communication protocols. Work has also been done on conversion of a LOTOS specification to C code [KvEvS90]. More recently, Straunstrup et al. [SAH+00] have described a new technique for evaluating large systems (incorporating a thousand or so concurrent components) in a matter of minutes; they use a tool similar to a model-checker to generate the states of a finite state machine.

We note, in passing, that the B specification language [Abr96], based on the Z specification language, resembles a programming language in its notation and use; hence the mapping and relationship between B and Z is quite strong. We believe this relationship to be the closest in the literature to the relationship between specification and implementation found in the CSP systems analysed in this thesis.

A similar tool to that of FDR is LCS [BS94], an experimental language aimed at exploring the design and implementation of programming languages based on CCS and CSP. The language extends Standard ML with primitives for concurrency and communication based

on the CCS formalism. As in our model, the abstract operational semantics are given in terms of a transition system. However, the major drawback with LCS, when compared with the FDR model-checker, is that explicit state and event information is not provided by LCS. Our analysis of divergence requires the state and event information which FDR provides. A survey was completed in the earlier stages of the research contained in this thesis to determine which model-checker was most applicable for our model; this led us to choose FDR.

Finally, BenAyed et al. [BDFM99] describe a number of techniques for deriving a mapping from specification to implementation. These include *software incrementation* (adding features to a software system in the same spirit as refinement) and *software adaptation* (modifying a program to satisfy a specification).

The chief difference between the related work just described and the research in this thesis is that we use the same language for specification and implementation; thus the transition from specification to final implementation is seamless. The drawback of the approach (if it can be considered a drawback) is the loss in flexibility of being able to choose the implementation language; herein, we are restricted to using CSP.

2.2.1 Process equivalence and refinement

The notion of the equivalence of two processes, not surprisingly, has received and is still receiving a lot of attention. Hennessy, de Nicola and Kennaway [Hen88] investigated various notions of testing a process. Two processes are said to be *equivalent* if they pass exactly the same tests. Although testing is not strictly related to the work in this thesis, some of the refinement checks undertaken in Chapters 6 and 7 are tests for certain conditions holding true. For example, in the case of the Towers of Hanoi problem, the terminating condition is

that the discs be properly arranged on pegs according to the rules of the puzzle. Hennessy [Hen88] describes two processes as having different behaviour if there is an experiment which one passes and the other does not. Milner [Mil89] describes two other types of equivalence. Two processes are *strongly equivalent* if both their internal and external actions are the same, while processes are *observationally equivalent* if only their external behaviour is the same. The notion of *hiding* or *abstraction* described in Chapter 4 permits internal behaviour to be *hidden*. In this case, we can have two processes which are observationally equivalent, yet not strongly equivalent. A good example of how this might occur can be found in Chapter 1, with the example of the CSP buffer implementation, which externally is equivalent to the specification, but has internally hidden behaviour making it not strongly equivalent. This leads us to the notion of *refinement* [WM90, AH93]: a process P refines another process Q if all possible behaviours of Q are possible behaviours of P. This implies that P could have behaviours (albeit hidden) which are not possible behaviours of Q. In this context, we can generalise refinement to specification and implementation.

The relevance and applicability of equivalence and refinement to our model is that we can view each *trace* of an implementation as equivalent to a *trace* in the specification, even though it may have internal (hidden) behaviour not included in its visible traces. In most cases, an implementation may refine the specification with additional behaviour. However, this additional behaviour of the refinement may be behaviour which is required in the implementation. Consequently, in this thesis, such behaviour is not considered as invalid behaviour, but part of the process of refinement. In the model used herein, we accept that an implementation will be observationally equivalent to its specification, but will contain additional behaviour also. Had we not accepted the possibility of hidden behaviour existing in the implementation, then a large number of the metrics we develop in Chapter 5 would not

have been collectable (central to the analysis of the four CSP systems in Chapters 6 and 7 is quantification of the difference between specification and implementation afforded by the metrics values we compute).

2.2.2 Divergence

The notion and measurement of the fidelity of software is inextricably tied to the concept of *divergence* [AH92]. According to Hoare [Hoa85], divergence is the general term for the phenomenon known in programming languages as an infinite loop. Milner [Mil89] defines a divergent agent as one having a cycle containing only the τ (internal) action. No other types of action are possible from then onwards, and since τ is internal, the behaviour of the agent is, observationally, that of an infinite loop. According to Roscoe [Ros94], a process is in a state of non-divergence if it is not divergent. In other words, there is an absence of a τ cycle. In our model, if a process is not divergent then it may be non-divergent or partially divergent. Brookes et al. [BHR84] take a similar view to Roscoe [Ros94]. A process is divergent if it is engaging in *infinite internal chatter*, or entails an infinite path all of whose labels are τ. The distinct difference between any other definition of divergence and our definition of divergence is the introduction of the intermediate state of partial divergence. We therefore extend the notion of divergence as it is usually thought of. The introduction of the state of partial divergence was considered essential for an analysis of CSP systems when they exhibit a state *between* that of non-divergence and divergence.

We cannot mention the area of divergence without some treatment of CSP and CCS. There is an intuitive relationship between our model and those of CSP [Hoa85] and CCS [Mil89], namely, that a non-divergent or partially divergent state in our model corresponds to a non-divergent state in [Hoa85] and [Ros94]. Most notable among the areas of CSP from which

we have drawn, and to which we added our own interpretation, is the CSP $\surd$ (tick) event. In a CSP sense, the $\surd$ event represents termination of the process which engages in it. The end of a trace is therefore signified by a single occurrence of the $\surd$ event. Since in CSP all processes are sequential, it becomes necessary to have an operator which signals the end of one process, and possibly the start of another. If a process P expressed in CSP becomes divergent (exhibits an infinite loop), then $\surd \notin P$ (i.e., the tick event is not an element of the set of events of the process). This is particularly important when composing processes in the form $P;Q$. Here, process Q begins when process P engages in the $\surd$ event. If $\surd \notin P$, then Q will never start executing.

In Chapter 4, we refine $\surd$ by introducing $\surd_{exp}$, representing termination with expected results, in which an end-state of the implementation is equivalent to that of an end-state in the specification. $\surd_{unexp}$ means termination with unexpected results, in which an end-state of the implementation is not equivalent to any end-state of the specification. From a theoretical point of view, the refinement of the $\surd$ event in terms of $\surd_{exp}$ and $\surd_{unexp}$ is a direct reflection of the fact that $\surd$ represents termination of a process, but makes no judgement as to the type or nature of the end-state. In our model, the presence of partial divergence is reflected in the values held by identifiers at a particular state. We also borrow the notion of a *trace* from CSP, but enhance its definition to include actions *and* states. The roles of hiding and relabelling, common to both CCS and CSP, are incorporated into our model in order to provide a mapping from specification to implementation.

2.2.3 Program correctness

Program correctness, or program proof as it is otherwise known, has received much coverage, and has stimulated widespread research over many years [LPP70, Gri81, Bac86]. The

problem of program correctness can be seen as comprising two parts. The first is *conditional* correctness (or *partial* correctness), and requires that a program be correct under the assumption that the program *does* terminate. The second, *total correctness*, requires that a program be correct and that the program *does* terminate. We can re-state these two concepts as follows:

1. *if* program X terminates, then S will be the set of results produced;

2. program X *does* terminate, *and* S will be the set of results produced.

Central to the idea of proving a program correct is the notation for propositions of the form:

```
{p1} st {p2}
```

where `p1` and `p2` are both propositions referring to identifiers owned by a proces P, say. The proposition can be read: if `p1` is true before statement `st` is executed, and `st` does terminate, then `p2` is true after `st` has executed. For example, the following axiom states that if, before the statement `x := x + 1` has been executed, `x` holds the value zero, then after executing that statement `x` will hold the value one.

```
{x = 0} x := x + 1 {x = 1}
```

We term `p1` the *pre-condition* on the action, and `p2` the *post-condition* on the action.

The above concepts establish the context of our work on divergence. We define our concepts of a non-divergent and a partially divergent state (Chapter 4) in terms of the values of identifiers held at each state. The notion of whether the identifiers at a particular state hold specified or unspecified values forms the basis of our decision as to whether a state is considered non-divergent or partially divergent. Since we view each trace that a process is capable of executing as terminating, our model is a model of total correctness; for now, we view a trace as simply an execution path.

The concept of program proof using the pre- and post-condition concepts was also adopted by Owicki and Gries [OG76] to allow concurrent programs to be proved correct but in a setting of a partial correctness model.

We can also define conditions which a program must fulfil in terms of its pre- and post-conditions, and then show how the associated code satisfies those conditions. One of the earliest program verifiers following this approach was that developed by King [Kin77], described at the time as a new approach to program testing. By carrying out refinement checks between specification and implementation, this guarantees, in the case of a true refinement, that every behaviour of the specification is possible by the implementation.

Quality factors

Since software fidelity is the theme of this thesis, our prime objective is to try and understand the relationship between a specification and its implementation, in terms of the emerging behaviour of the latter through refinement and to develop a set of metrics applicable to all CSP systems. Fidelity of an implementation is a factor in the quality of a system. We thus need to relate this quality factor to other indicators of quality.

Laprie [Lap95] defines (system) dependability as the reliance which can justifiably be placed on the service provided by a computer system; this is a definition echoed in [LA90]. Dependability can be seen in terms of *viewpoints* depending on the part of the system under consideration. For example, readiness for usage in terms of its availability; continuity of service in terms of its reliability; the non-occurrence of unauthorised disclosure of information in terms of its integrity, etc. System attributes such as availability and reliability, according to this definition, become facets of dependability. Of interest is a paper by Waeselynck and Boulanger [WB95], in which the B pseudo-specification language [Abr96] is used as

a framework for stringent testing of a live system. Therein, elements of testing, formal development, reliability and dependability are encapsulated.

2.2.4 Model-checkers

Related to the area of program testing and program proving are a number of *model-checkers*, most of which take a representation of a specification and its implementation and allow a comparison of the two to be made. Key to the comparison of specification and implementation in Chapters 6 and 7 is the use of the FDR model-checker. A variety of other model-checkers are in general use undertaking similar tasks to those of FDR.

The SPIN model-checker [Hol95] allows specific properties of a process to be proven as holding true under all executions of that process. Within SPIN, a model is specified using the language PROMELA, incorporating conventional program constructs, and allowing the specification of channel variables. A simulation of program execution is then performed, and C code is generated to perform a validation of the program state space. It therefore encompasses elements of exhaustive testing and combines this with a feature for proving properties of a process correct. In providing these features, the designer of SPIN accepted that proving every property of a program under all conditions was too large a task to be undertaken. Dillon [DY94] incorporates the use of test oracles (in a similar fashion to SPIN) in order to prove temporal properties of processes. A GIL (Graphical Interface Logic) in which properties of processes can be proved and shown graphically is used to perform tests establishing properties which hold during specific time intervals. This approach combines elements of testing theory and program proving, and again accepts that to show all properties of a process as holding true is too sizeable a task.

Of interest in tying the two fields of probability and model-checking is the work done

in producing PRAVDA [Low91], a tool developed for verifying probabilistic communicating processes. A specification and its implementation are taken as input, and a probability as to whether certain events are likely to occur can be given. For example, given a process which is capable of executing an event α, but is dependent on event β occurring first, then a probability that event α will occur within a specific time frame can be estimated.

In the following section, related and complementary work in the field of software metrics is described.

2.3 Software metrics

A software metric, or simply metric, can be defined as any quantifiable measure of the behaviour and characteristics of a software system. As a measure of software attributes, software metrics play an important role in the field of software engineering. Better understanding of the development process, sound software design principles and the ability to better estimate costs and effort of future projects are just a few of the potential benefits of collecting and using metrics [Gil77, FP96, She95]. Software metrics can help us to identify and understand various features of software products and processes. For example, finding a relationship between the number of faults found in a program module and the coupling of the module (in terms of associations with other modules) can help us understand what constitutes a generally recognised optimum coupling level. This optimum level can then be used to assess whether, in future projects, the right amount of coupling has been used. In Chapter 5, we propose a set of metrics for capturing attributes of CSP systems. Thus, we are able to, firstly, establish features common to certain types of problem domain, and, secondly, establish differences between types of problem domain.

As we noted in Chapter 1, some of the metrics we proposed proved useful in revealing

features of Type I and Type II CSP systems. Others were found to be inappropriate for the task, and in that case we trimmed and added to the original set of metrics. In order for a software metric to have any real value therefore, it must be appropriate in the sense that it is useful and meaningful for the problem it is applied to. Hence, the theoretical considerations for the validity of metrics are as important as the empirical evaluation of metrics. This evaluation may be conducted through repeated experimentation (or, as in the case of this thesis, through repeated running of refinement checks with an increasing cardinality of the state space). The theoretical approach to the validation of metrics requires us to clarify what attributes of software we are measuring, and how we go about measuring those attributes [Fen94, KPF95, BBM96]; a metric must measure what it claims to measure. Kitchenham et al. [KPF95] describe a list of features for a metric which must hold for that metric to be theoretically valid. In Chapter 5 we address some of the theoretical issues involved in the development of the proposed set of metrics, and so we postpone a full treatment of these issues until that point.

Fidelity and quality

Throughout this thesis, it is important that we do not lose sight of the high-level objectives stated in Chapter 1. We must also retain a hold on why fidelity is important and understand what is different about fidelity when compared to other quality features. After all, improving the quality of software is the underlying aim of all software engineering practice [Som98].

Herein we view the quality factor of fidelity in terms of the extent to which an implementation refines a specification. This is very different to refinement itself. The notion of fidelity is the measurement of the process of refinement rather than the act of merely doing the refinement itself. Fidelity would thus be the top-level feature we would be trying to capture

through measurement of the refinement process. At lower levels, the notion of fidelity is expressed in terms of the states, actions and other relevant features of a process at different stages of refinement, in other words, the proposed metrics. Differences between the states, actions and these metrics of the specification and implementation are then quantifiably comparable and hence can be used to get a clearer picture of the nature of the high-level quality factor of fidelity. Development of a quality model is an essential step in clarifying what we are attempting to measure from a high-level abstract perspective right through to low-level countable measures.

Many models have been suggested as a means of identifying quality. Most notable amongst these has been the GQM (Goal Question Metric) method [BR88]. The metrics proposed in this thesis were developed along similar lines to those of the GQM approach. There are also quality guidelines based on the ISO 9126 standard [ISO91] and the QMS sub-system [KWD86], which identify the individual quality factors making up the overall view of quality. For example, in the ISO 9126 standard, *reliability* has the criteria of: availability, correctness and fault-tolerance; these criteria are then broken down further and the metrics are computed at the lowest level of the resulting tree-like structure. Availability, for example, is broken down further into directly measurable attributes such as: percentage of time machine is available over a specific time period, response times and maximum loads, etc.

Fenton and Pfleeger [FP96] describe a number of techniques and practices for producing software metrics. This includes a description of cost models such as COCOMO [Boe81], and statistical tests and experiments which can be applied to software.

In our analysis, which incorporates two types of state, we have chosen metrics which best reflect the level of refinement between a specification and its implementation; the proposed

metrics emphasise states, actions, hidden behaviour and sub-processes in the specification and implementation. We do not claim our metrics to be any more valid than any other proposed to date for measuring software. For example, we could assess the quality of a system in terms of a metric capturing the features of implementation time and effort, its work capacity, and overall costs. Standards could then be set for achieving certain goals related to these three features. We could then define *acceptable* and *unacceptable* levels of quality. The important point to note about the set of metrics we propose in Chapter 5 is that they give us a means by which we can evaluate the behaviour of CSP processes, even if they only apply to certain types of CSP process (which, as it turned out, was indeed the case).

Other metric suites

Various object-oriented metrics suites have been suggested as a means of determining whether the systems under investigation hold desired properties of object-oriented software, or whether that software is of sufficient quality. The MOOSE (Metrics for Object-Oriented Software Engineering) set of object-oriented metrics started a sequence of empirical studies into the features of the Smalltalk and C++ object-oriented languages [CK94]. Other sets of metrics have also tackled the object-oriented paradigm [Lor93], and it is only, now, after a considerable number of experiments, case-studies, etc., that the object-oriented community has begun to assess issues relevant to the paradigm. Only now are we beginning to see the real problems faced in the OO world. In terms of this thesis, the empirical study carried out (Chapters 6 and 7) only really scratches the surface of analysing the behaviour of CSP processes during refinement. One of the problems of doing such research is that it reveals more questions than it answers. Future work (Chapter 8) does, however, suggest a number

of alternative ways in which the study can be taken further.

The nature of our metrics

To understand the nature of our metrics, it is important to make the distinction between a *product* and a *process* metric. A product metric is a measure of a software artifact, for example, a piece of code. It relates to the product itself, and as such is a metric of a static artifact. A process metric is a measure of the process of software development rather than the product itself; for example, the number of hours spent carrying out software maintenance or development. A similar approach to the comparison of specification and implementation (as we have done in this thesis), in which product and process metrics play a part, is found in the Balboa tool [CW98]. Balboa is a process validation tool which allows discrepancies between static process models and their execution equivalents to be assessed. Balboa allows the capture of event process data, where *event data* refers to actions performed by agents, whether they be a human or an automaton. An event characterises the behaviour of a process in terms of identifiable, instantaneous events. It therefore takes a behavioural approach to the study of processes. Metrics measuring the discrepancies between the static process model and its execution equivalent are proposed and evaluated.

We view processes as combinations of states and actions. A set of refinement metrics using our definition of fidelity could be seen as both product and process metrics; they are *hybrid* metrics, since they chart the progress of an implementation from a high-level abstract design to a low-level implementation (i.e., mapping the process of development through a number of stages). At each stage the metrics are reflective of static features of the process (in terms of states, actions and associated metrics) and hence are, in a sense, product metrics as well. The artifacts under study at each stage are the CSP constructs of the specification

and its implementation.

The fact that our metrics are product and process metrics combined distinguishes our work from other studies which tend to focus on either product or process metrics [McC76, Hal77, YW78, Hum90], but not metrics which are a combination of the two. Also of relevance are the current efforts aimed at improving our understanding of techniques for estimating software process characteristics [SC01, SK01].

2.3.1 Empirical software engineering

Empirical software engineering can be defined as the study of software-related artifacts for the purpose of characterisation, understanding, evaluation, prediction, control, management or improvement through qualitative or quantitative analyses. Empirical research within the software engineering arena can be used to investigate the association between proposed software metrics and other indicators of software quality such as maintainability or comprehensibility. Qualitative or quantitative analyses, through the use of metrics, can be used to support these investigations [Sch92, BBM96, BDM97].

As well as being theoretically valid, it is also useful to have a metric that can be shown to be of use through empirical evaluation. For example, an empirical investigation which attempts to identify design metrics most useful to the software engineer has been described in [IS89]. A metric based on information flow is introduced, and found to correlate highly with development effort. In the same study, various code metrics are shown to be poor indicators of development effort. The numbers of events and states found in software systems were also investigated in [CS00] and shown to be useful indicators of system features (as we also found in our analysis in Chapters 6 and 7).

The careful planning of an empirical study is essential to the success of any study. Careful

planning of a study also means that further tests can be undertaken and the study can be replicated by other researchers; this is important for hypotheses to be confirmed or refuted. Herein, accurate collection of data and accurate dissemination of information and reporting of results to the software engineering community was also important for further studies to be carried out effectively. In the analysis of the four CSP systems (Chapters 6 and 7), the data, i.e., the metrics' values, are collected automatically by software and then analysed using a spreadsheet. Future work will attempt to provide automatic analysis from the collected data.

In a paper by Votta et al. [VPP95], a number of current problems in the experimental software engineering world are described. These include the suggestion that repeated experiments are not valued as important research, that a poor synergy currently exists between computer science, software engineering and software development enterprises and that too many proposed theories in the computer science world cannot be tested, and are, hence, of little use to the empirical community. A number of suggestions are made to improve the current situation in the empirical software engineering community. These include the need to repeat experiments, and the need for access to real project data in order for the community as a whole to move forward.

Herein, we have tried to create the conditions for the same type of test to be repeated and as far as possible to use real systems developed by CSP developers. As an epilogue we state and answer five questions, originally proposed by Fenton [Fen94], pertaining to any claim made of software engineering research:

1. Is it based on empirical evaluation of data?

2. Was the experiment designed correctly?

3. Is it based on a toy or real situation?

4. Were the measurements used appropriate?

5. Was the experiment run over a sufficient period of time?

In terms of the research herein, Question 1 is certainly true. Data, in the form of metrics values, was collected and analysed using a spreadsheet. Although the evaluation in Chapters 6 and 7 could not really be seen as constituting an experiment, but more of a case study approach, it is nonetheless true to say that the number and types of refinement checks undertaken were thought through very carefully prior to them being run (Question 2). Every effort was made to extract the most information from the planned refinement checks, even though in certain cases that information was not what we had hoped for. The CSP systems analysed herein are not toy problems, certainly not in terms of the cardinality of the state spaces of the two Type I CSP systems. They were written by developers familiar and experienced with the CSP paradigm (Question 3). We have to admit that, on reflection, a number of the measurements (metrics), initially thought to be useful, did not turn out to be as appropriate as we had hoped (Question 4). Finally, Question 5 relates mostly to experimental conditions, and hence it is difficult in this particular case to answer.

2.4 Summary

In this chapter, we have described related and complementary work, undertaken in the areas related to our model of fidelity. Some areas have direct relevance and inform our model. Others have less influence, and some are described as a way of comparing a completely different approach to ours. The main areas described were formal methods, incorporating process equivalence, divergence and program correctness and software metrics, on which

our empirical analysis rests; incorporated within software metrics is the area of empirical software engineering.

What we have described, therefore, are the major influences in the way our model has been shaped and developed, how some areas have been complemented by our approach and why we decided upon the research direction we did. In the next chapter, we begin a formal description of some of the techniques we have just described and, in particular, the fundamental concepts upon which our model is built, namely, that of state, action, finite state machine and labelled transition system.

Chapter 3

Foundations of a divergence model

3.1 Introduction

In chapters 1 and 2 we have given an informal description of our interpretation of divergence, and how this augmented our definition of a non-terminating program. The objective of this chapter is to place our understanding of divergence and its theoretical foundations on a more formal footing.

According to Brookes [Bro83] and Hoare [Hoa85], an arbitrary process X is capable of exhibiting the phenomenon of *divergence* if it is capable of performing an action an infinite number of times. This can be any action within the set of possible actions a process is capable of engaging in. No claim is made as to whether this is how we expect process X to behave, or whether or not this is a desirable behaviour.

Henceforth, we describe an executing process as *divergent* if it was intended to terminate but is currently exhibiting characteristics of non-termination. We describe as *non-divergent* any process which we expect to terminate, and which is not currently exhibiting the characteristics which would indicate that it was a non-terminating process. For such a process,

this is either because it will eventually terminate, or because it has not yet reached a point in its execution where the characteristics of non-termination are observable. In what follows we use the term *status*, as distinct from *state*, to describe whether a process is currently divergent or non-divergent.

We can quite easily envisage a scenario in which a process begins its execution as non-divergent, and becomes divergent at a later point (from then onwards, the process is incapable of returning to a non-divergent status). If we are to view each of the two extremes of non-divergence and divergence as a possible status of a process, then there lies an intermediate status of a process characterised by a sequence of actions which take that process from a non-divergent status to divergent status; henceforth we find it convenient to view this very sequence of actions as also defining the status of a process. In saying this, however, we are not implying that every process which exhibits the characteristics of this intermediate status will necessarily eventually become divergent.

Herein we introduce the theoretical foundations upon which our analysis of divergence is based by employing Finite State Machines (FSMs) and Labelled Transition Systems (LTSs).

In Section 3.2 we discuss the concepts of an *action* and a *state*. We give an example to illustrate the importance of these two concepts. In Section 3.3 we explain the need for a formal definition of an FSM and give an example to illustrate how the principles of an FSM apply in the real world. In Section 3.4 we define a (deterministic) FSM; we also introduce the idea of a Communicating Finite State Machine (CFSM), capable of a restricted set of actions. In Section 3.5 we introduce the concept of a labelled transition system, showing how it can be used to define a process in terms of traces. We then give some examples to illustrate these definitions. The chapter is summarised in Section 3.6.

3.2 Preliminaries

We use the term *specification* as defining the intended behaviour of an *implementation*; an implementation is an attempt to realise that specification. Given these informal definitions, we consider two different definitions of software non-fidelity, namely either:

- the specification is met, but that specification is incorrect (the specification does not adequately capture the user's real requirements),

or, alternatively:

- the specification is correct (the specification does capture the user's real requirements), but the implementation exhibits unspecified behaviour.

In this and subsequent chapters, we use the latter definition of software non-fidelity[1]. We are then in a position from which we can make appropriate comparisons between the behaviours of different implementations. In order for the last claim to have any meaning, we must first show that there is an intuitive and physically realisable relationship between a specification and its implementation. Only if we can show a level of commonality in terms of the behaviour of a specification and its implementation can we hope to compare the two.

We can think of a specification as expressing *what* functions a process should perform but not *how* it should perform those functions. We assume the existence of a (multi-valued) *mapping function* from the specification to a number of implementations each of which attempts to meet that specification. (In O'Neill [O'N92] it is shown how a mapping between VDM [Jon86] and SML [Mye79] is accomplished.) If none of the behaviours of an implementation is a behaviour of its specification, then that mapping fulfils no function whatsoever, since

[1] The former interpretation is the subject of current *requirements analysis* research, and as such is beyond the scope of our work. The interested reader should consult Somerville [Som98].

there is no correspondence between the two. At the other extreme, if neither specification nor implementation is capable of exhibiting behaviour not possible of the other then their behaviours are equivalent. In that case, we find a correspondence between every behaviour of the specification and every behaviour of its implementation.

Until now we have talked about the status of a process as being either non-divergent or partially divergent. To avoid confusion we have deliberately avoided use of the term *state*. We want to reserve its use for a more formal discussion.

For the moment, if we view a *state* as a particular configuration in the store of a computer, then details about certain states in the implementation would also be a necessary part of the specification.

In general, given a specification S, let I be a concrete implementation of S. Included in I would be the identifiers, actions and states necessary to demonstrate the equivalence of I and S. In practice, the *granularity* of an implementation I when compared with a specification S might vary greatly, i.e., the extent to which S would have to be refined to yield I. An implementation is unlikely to comprise exactly the same number of identifiers, actions and states as its specification. The problem is therefore one of finding a way in which a specification can be compared with an implementation, irrespective of the granularities of either. Fortunately, the proliferation of tools currently available on the market for choosing the granularities of specification and implementation makes our task easier. For example, we could choose VDM as the specification language and SML as the implementation language; environments exist already and research is still being undertaken in mapping between various specification and implementation languages (see [O'N92], [AM94] and [AH00]). However, in any comparison we might hope to make between a specification and its implementation, we need to define the underlying models we intend to use as a basis for that comparison.

3.3 The need for formal definitions

We are interested in using a specification language and an implementation language which give us as close a mapping between that specification and implementation as possible [2]. It follows that the closer the mapping we can obtain, the easier will be the judgment on their equivalence or non-equivalence, and the easier we can obtain information on the differences between specification and implementation [3].

Fundamental to the idea of expressing specifications and implementations in such a way that the said mapping can be made trivial is the mathematical concept of an FSM. Before giving the formal definition of an FSM, we describe how, additionally, the notion of a model incorporating actions and states is appropriate in the physical world. To illustrate this point, we consider the following example.

Example

A sweets vending machine is being used by a customer, in which each *action* by that customer is reflected as a *state* of the vending machine following that action. We assume, for the sake of clarity, that no error conditions arise in this transaction, i.e., the vending machine behaves correctly on every occasion. The action by the customer of *insert-coin* will yield a vending machine state of *coin-inserted* and the action by the customer of *make-selection* will yield a vending machine state of *sweet-dispensed*.

We can show this sequence diagrammatically in Figure 3.1, in which the start of the transaction is given by state q_0 (which is also the final state of the transaction). State q_1

[2] The disparity between a specification language and an implementation language is known as an *impedance mismatch*.

[3] In Chapter 6 we discuss a tool, called Failures-Divergence Refinement (FDR) [For97], based on the theory of Communicating Sequential Processes (CSP) [Hoa85], in which a specification and an implementation are expressed using the same medium.

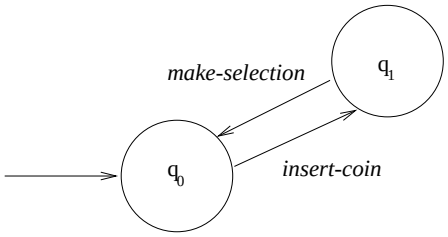


Figure 3.1: A vending machine with states and actions

represents the coin-inserted state of the vending machine, and the actions are labelled on the arrows (or *arcs*).

The vending machine, under normal operation, has specified states and actions, and provides us with an idea of how actions and states are inextricably linked; this is a theme we will return to at length in later chapters. We next define formally an FSM; this will form the theoretical basis of the definitions of a non-divergent and partially divergent state which we will give in Chapter 4.

3.4 Finite state machines

Thus far, we have viewed a state as a particular configuration in the store of a computer. We took this notion of a state to include the *implicit* and *explicit* addresses in store, the former relating to the contents of registers in a physical store (e.g., program counter), the latter relating to the contents of user program identifiers. The ensuing definition, taken from Cooke and Bez [CB84], marks the start of a formalised approach to the notion of divergence.

Definition 1:

A deterministic finite state machine (FSM) M is an algebraic structure:

$$M = (Q, \Sigma, t, q_0, F)$$

where

Q is a non-empty finite set of *states*

Σ is a non-empty finite *input alphabet* (or set of actions)

t is a mapping $Q \times \Sigma \longrightarrow Q$, called a *state transition function*

$q_0 \in Q$ is the initial state

F, $F \subseteq Q$, is the set of *final* states (or *accepting* states).

(For the sake of clarity, we omit the usual *print* (or *output*) function p defined by p: $Q \times \Sigma \longrightarrow \Sigma$. In some cases, we distinguish between an input alphabet (Σ) and an output alphabet (Σ') to indicate that they need not be identical[4].)

The behaviour of M can be shown diagrammatically via a directed graph, in which states are represented by labelled circles, and elements of F have a further circle drawn around them. We capture the essential characteristics of a finite state machine in Figure 3.2, where the input alphabet is $\{a, b\}$. The finite state machine is capable of accepting the sequences:

$$\langle\ a, b\ \rangle,\ \langle\ b, a \rangle,$$

the empty sequence $\langle\rangle$, and any sequence obtained by a concatenation of these three alternatives any number of times.

3.4.1 State transition notation

Since the idea of a process consisting of a series of state transitions occurs frequently in this and future chapters, we find it convenient to adopt a notation in which we can express

[4]This machine is called a *finite state transducer* [CB84].

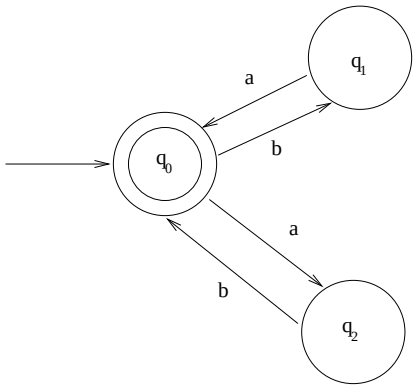


Figure 3.2: A finite state machine

transitions simply; we represent a single state transition using the notation

$$s \xrightarrow{a} s'$$

(or (s, a, s')) to describe a transition from state s to state s' by virtue of action a, where

(1) $s, s' \in Q$

(2) $a \in \Sigma$ and $t(s, a) = s'$.

Here t maps a given *source* state s and an action a to a *target* state s'.

In Figure 3.2, we assume the actions to be *read* actions, capable of reading a or b. As a result, possible types of action include:

- a *read* action drawn from an input alphabet which we could label Σ_0, e.g., $read(x)$ reads a value into identifier x from a source external to the program.
- a *write* action drawn from an alphabet of actions which we could label Σ_1, e.g., $write(x)$ writes out the current contents of identifier x.
- an *assignment* action in the usual sense of an assignment, e.g., `x := y - 1` sets the

identifier `x` to be one less than identifier `y`; we could label this alphabet Σ_2.

We could then amend the transition function t to incorporate the above scenario (t is now a mapping $Q \times (\Sigma_0 \cup \Sigma_1 \cup \Sigma_2) \longrightarrow Q$) giving us a certain flexibility in defining a non-reading action. To illustrate the importance of having this flexibility, we could draw the underlying graph of a *communicating process*[5] showing how it comprises only read and write actions. Any underlying model for such a communicating process would have to incorporate both types of action (*read* and *write*) within its alphabet. As a further illustration of this flexibility, an imperative high-level language would have actions for reading, writing, assignment etc., and we could quite easily model any finite number of types of action within a single FSM. For the moment, we retain just read and write actions in our definition of an FSM, on the understanding that each high-level action can be expressed in terms of just these two *primitive* actions.

As an example of the translation of a high-level imperative language, complete with added concurrent constructs, into a form where the semantics of that language is captured by the above primitive actions, see Milner [Mil89]. In the example which follows, we use the notation of *Communicating Sequential Processes* (CSP) to describe a simple copy process and show the similarity with a single high-level assignment command. For further information on the CSP notation we refer the reader to Hoare [Hoa85].

Example

The high-level assignment `x := 2` can be represented by a CSP process which offers a value `2` on a channel `left`, and places the value of `x` on a channel `right`. The two respective actions are written:

[5] A *communicating process* is one whose source of input is data passed to it on channels by other communicating processes to which it is connected; its output is directed to those other processes.

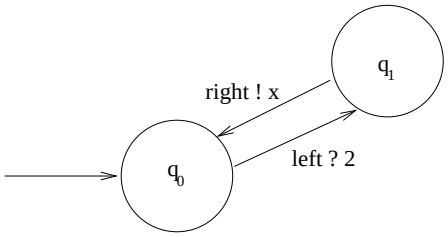


Figure 3.3: Finite state machine of a communicating process

Figure 3.4: Block diagram of *CSPPROC*

```
left ? 2      right ! x
```

We can represent this process by the finite state machine shown in Figure 3.3, and in CSP notation (where the notation `->` can be read as: '*followed by*'):

```
CSPPROC = left ? 2 -> right ! x
```

Informally this can be stated as:

> *The left channel offers the value 2 to the right channel. When the exchange of the value 2 has been made, x holds the value 2*

and is shown diagrammatically in Figure 3.4. The acceptance of the value on the right channel is synchronised in the absence of buffered channels; the `left` and `right` channels may be linked, but we have not incorporated these features into Figure 3.4. The last example exposes a major drawback of our current definition of a finite state machine so far. It fails to consider the concept of *channel variables* holding values, e.g., `left ?  2` indicates that the `left` channel is willing to accept the value `2`. An ability of our model to incorporate the concept of channel variables holding values will prove an important feature of our approach to divergence.

3.4.2 A communicating finite state machine

In view of the preceding discussion, we would then be justified in *extending* our basic FSM to form a *communicating* finite state machine (CFSM) [Hol91, Hol], defined as a 5-tuple (Q, A, t, q_0, M), where A is a set of variable names (i.e., channel names – `left`, `right`), M is the set of all *message queues*[6] currently on the named channels of a process (we will assume the queue to be of length greater than one), and Q, q_0 and t retain their definitions as before; A replaces Σ. Elements of the set M could represent the current state of a process, expressed in terms of the values currently held on all the channels it uses. For our own purposes, we re-define A (the alphabet of the process), namely, it now comprises all different combinations of channel names and values which can validly be held on those channels; alphabet A would therefore be a mixture of *read* and *write* actions. We could have incorporated a set of final states into our definition, but prefer to assume that a CFSM is a continuously operating process, without any valid final states. Finally, a message queue is taken to consist of:

(1) the number of slots in the queue (we have assumed at least two);

(2) the current queue contents.

3.5 Transition systems

For a theoretical underpinning, FSMs provide a useful starting point. However, in order that we can give further meaning to the concepts of action and state, we must look to the notion of a *Labelled Transition System* (LTS). As in the testing methodology of Hennessy [Hen88], using an approach based on LTSs will allow us to define our model in terms of its

[6] We can think of a message queue as a buffer of messages waiting to be processed.

operational semantics, a characteristic not afforded us by a treatment that solely relies on FSMs.

Central to the theme of processes being characterised by their actions and states is the notion of a *Transition System* incorporating the possible states of that process, and the state changes (or *transitions*) the process is capable of making. We first consider a simple transition system [Arn94] in which we view transitions and states as unsequenced. Hence, there are two mappings for each transition.

Definition 2:

A *transition system* is a quadruple $\mathcal{T} = \langle Q, T, \alpha, \beta \rangle$ where:

Q is a finite or infinite set of *states*

T is a finite or infinite set of *transitions*

α and β are two mappings from T to Q mapping each transition t in T to the *source* ($\alpha(t)$) and *target* ($\beta(t)$) states, respectively.

We next consider the definition of a *path*, taking into consideration the sequence of states.

Definition 3:

A *path* of finite length n ($n \geq 1$) in a transition system $\mathcal{T}$ is a sequence of transitions $t_1, ..., t_n$ such that $\forall i : 1 \leq i < n$, $\beta(t_i) = \alpha(t_{i+1})$. A path of infinite length can be similarly defined.

3.5.1 Labelled transition systems

We now refine further the definition of a transition system to incorporate an *alphabet* of actions, reflecting possible actions the transition system is capable of executing. This is known as a *labelled transition system* (LTS). In the sequel, we can think of a *label* as being equivalent to an action.

Definition 4:

A transition system *labelled* by a finite alphabet Σ is a 5-tuple $\mathcal{L} = \langle Q, T, \alpha, \beta, \lambda \rangle$ where

$\langle Q, T, \alpha, \beta \rangle$ is a transition system; and

λ is a mapping from T to Σ ($\lambda : T \longrightarrow \Sigma$) taking each transition t in T to its label $\lambda(t)$.

We note that an LTS is essentially a non-deterministic FSM with no specific initial or final states.

We now define the concept of a trace, whose importance in our model should become evident as we develop the model further. In the sequel Σ^* denotes the Kleene operator over Σ.

Definition 5:

A finite *trace* of a path $t_1, ..., t_n$ is the sequence $\lambda(t_1), ..., \lambda(t_n)$ in Σ^*.

Definition 6:

An infinite *trace* of a path $t_1, t_2, t_3, ...$ is the sequence $\lambda(t_1), \lambda(t_2), \lambda(t_3), ...$ in Σ^*.

We are now in a position to define a process in terms of an LTS.

Definition 7:

A *process* is an LTS with a specific initial state.

We consider an implementation process as comprising one or more traces. If the specification process is expressed in the same way, we also view it as comprising one or more traces.

Example 1

A sweets vending machine has a potentially infinite set of traces of finite length, where a trace equates to the actions and states of a single customer transaction[7]. The alphabet of actions contains the valid customer and vending machine actions, the states of the vending machine being determined by those actions. For example, a single trace of `insert-coin` followed by `make-selection` for a sweets vending machine process `SVM` could be expressed as a sequence whose elements are of the form (s, a, s').

```
SVM = <(no-coin-inserted, insert-coin,    coin-inserted),
       (coin-inserted,    make-selection, sweet-dispensed),
       (sweet-dispensed, set-to-ready, no-coin-inserted)>
```

Here, the `set-to-ready` action is automatically executed by `SVM` after dispensing a sweet, i.e., an indicator light shows its readiness to accept another coin as part of a new transaction.

[7] In practice, the set of traces is finite, as eventually the vending machine becomes obsolete and is replaced.

Example 2

In CSP, the process STOP has a single trace, namely, that of the empty sequence $\langle\rangle$. This would not be a suitable process to model a vending machine, since it would be capable of no actions, and would yield no states, apart from that given by its only state, the *null* state.

Example 3

Consider the example of a process `COUNTDOWN` which takes a value `x` between one and five, and applies the action `dec` (which decrements by 1) until `x` reaches zero, at which point its single trace terminates. The states of `COUNTDOWN` are the values of `x` at any moment, and the labelled action is simply `dec`. We ignore the possibility of error conditions (e.g., `dec 0`) in order to illustrate the point being made.

This gives us a transition system for `COUNTDOWN` capable of a finite trace, with the following possible transitions (in the format (s, a, s')), extended to become a labelled transition system with a single label (or action) `dec`.

```
(5, dec, 4)

(4, dec, 3)

(3, dec, 2)

(2, dec, 1)

(1, dec, 0)
```

We include the states s and s' for each action a on the trace:

```
{<>,

<(1, dec, 0)>,
```

```
<(2, dec, 1), (1, dec, 0)>,

<(3, dec, 2), (2, dec, 1), (1, dec, 0)>,

<(4, dec, 3), (3, dec, 2), (2, dec, 1), (1, dec, 0)>,

<(5, dec, 4), (4, dec, 3), (3, dec, 2), (2, dec, 1), (1, dec, 0)>}
```

The above examples give us just an intuitive feel for the notion of a state, in terms of FSMs and LTSs. So, thus far, we have only hinted at a possible definition of a state, cast in terms afforded by an LTS (see Section 4.4 for the details).

3.6 Summary

In this chapter we have laid the theoretical foundations of a model which we will be building on in subsequent chapters, and will use for our analysis of divergence in processes. This was accomplished via the concepts of an FSM and an LTS. We have demonstrated how an LTS gives us greater flexibility in our definition of a process as well as the traces which make up that process.

In the next chapter, in addition to formally defining the notions of our two different types of state, we pause to examine some alternative definitions of the concept of a state in order to illustrate that there is no single definition which can be generally applied. The definition of a state varies from model to model, and from application to application. For example, models with temporal or timed aspects to their definition must incorporate the notion of physical time into their notion of a state [Sch90]. Strain-Clark et al. [SCMC94] describe the state of a process as only the mode of system behaviour which is externally observable, whilst Abowd [AD94] describes a state as the internal information of an entity, and reserves the term *status* for the externally available information about that entity. In the ensuing

chapter, we also examine the inter-relationships between actions and states.

Chapter 4

Divergent states and actions

4.1 Introduction

The objective of this chapter is to define the notion of the state of a process. We also define non-divergent and partially divergent states. We extend the notion of divergence to traces and processes and consider the role of actions in the context of divergence. The two types of state, together with the actions a process engages in, form the foundation of our model and the basis of our analysis and evaluation in later chapters of the thesis.

In Section 4.2, we review some definitions of state in the literature and give an example which anticipates our definition of state. In Section 4.3, we describe the difficuties in trying to compare a specification and its implementation. In Section 4.4, we formalise the notion of divergence allowing us to provide definitions of a non-divergent and a partially divergent state, together with some other properties of processes which will prove useful in Chapters 6 and 7. In Section 4.5 we analyse, with examples, the impact an action can have on the state of a process and, as a result, identify features of CSP which will prove useful in later chapters. These features are formalised in Section 4.6 and a summary is given in Section

4.2 Review of state definitions

According to Milner [Mil89], expressions relating an *agent* (a system entity capable of performing actions) to the current values of those expressions can be considered the *state* of that agent. The state of a system is then the combination of the states of the individual agents.

According to Roscoe [Ros94], in the context of a CSP model-checker, the state of a process can be viewed in terms of the values of parameters currently being passed to that process. For example, the state of a recursively defined factorial function is defined in terms of the partial results of that computation.

According to Holzmann, in the context of SPIN [Hol95], a model-checker for verifying properties of concurrent systems, a system consists of three types of object each of which has a local state. These are summarised in Table 4.1.

Object Type	*Local State*
Data	Value
Message Channel	Contents
Processes	Program Counter (and local objects)

Table 4.1: Objects of a system

According to Hoare [Hoa81], the state of a *message channel* is the sequence of messages that have passed through the channel and the set of messages that can enter it. The state of the overall process is the combined states of all message channels used by that process. Two states s_1 and s_2 are then equivalent if every executable sequence of actions starting from s_1 leading to a state at which a set of actions X are executable is also executable starting from

s_2 and leading to a state at which the same set of actions X are executable.

Each of the above state definitions emphasises the values held by data objects. In the next section we illustrate, with examples, the difficuties in achieving an equivalence between the states of a specification and its implementation. This will pave the way for a formal definition of the two types of state of our model.

4.3 Difficulties in achieving equivalence

Consider the pseudo-code of Example 1 which we assume, for the moment, to be the specification of a process. In reality, a specification is unlikely to contain the details required for an implementation to be coded verbatim. We use the example of a sequential program to illustrate the principles involved.

Example 1: a sequential program

We assume that `x` and `y` are identifiers for integer variables, to which the values `2` and `4` are assigned, respectively. (We assume a correct initial state q_0 to be one in which `x` and `y` are undefined.)

```
begin
    y = 4
    x = 2
end.
```

We could equally have expressed these requirements in the VDM specification language[1], and converted that specification to SML in the spirit of [O'N92]. We can annotate this

[1] In VDM, this would have appeared as the *definition* of two *values* `x` and `y`.

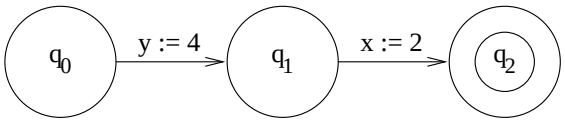


Figure 4.1: Finite state machine of a sequential program

specification with the expected values of the variables `x` and `y` after each action, and at each state. These annotations can be seen as *pre-* and *post-conditions* of the action [Hoa69]; we enclose those expected values between {}. We can annotate the implementation in the same way (the implementation could represent the SML produced from the VDM specification[2]). We assume this implementation matches the specification line for line, giving:

```
begin
{x, y undefined}
  y = 4;
{y = 4, x undefined}
  x = 2;
{y = 4, x = 2}
end.
```

We can represent this implementation by a transition system with three states, and two actions, as shown in Figure 4.1.

This example illustrates how, by annotating both specification and implementation, we gain insight into how we might compare the two and analyse their differences. However, we have engineered this example so there is an exact match between the actions and states in the specification and implementation. This is an unrealistic assumption for three reasons:

1. A single action in the specification may be implemented by n actions in the implementation or vice versa.

[2]In SML, we would have to use *reference* variables, considered non-standard in SML and most functional languages.

2. There may be identifiers in the specification for which there are no corresponding identifiers in the implementation or vice versa.

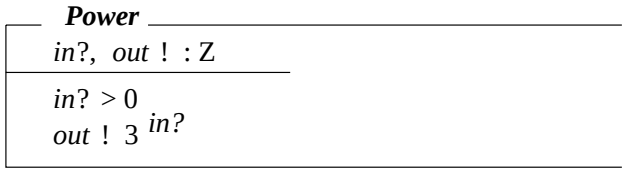
3. When using a high-level specification language such as Z, a specification defines what the program should do, not how it should do it. This means there is unlikely to be a simple mapping betwen the two. Therefore, establishing a correspondence between the states and actions in the specification and implementation would prove difficult in the majority of cases.

In the next example, we illustrate the last of these problems using a power function specified using the Z specification language.

Example 2: power function in Z

This example is based on one given by Diller [Dil95] which raises three to the power of *in*.

First, we describe the notation used. In a Z specification, operations are specified in terms of input-output behaviour. The operations are packaged into a *schema*. Every schema has a name placed at the top of the schema box. The *declaration* part is written below the name of the schema, followed by the *predicate* part of the schema, as shown below for the *Power* schema.



The function specifies in the declaration an input value *in*? and an output value *out*!, both of which are integers. The predicate part of the schema has two elements. The condition on the input value is that it has to be greater than zero. The output value is three raised to the

power of the input value. The program code for such a specification could be implemented recursively or iteratively. The following code shows a recursive implementation:

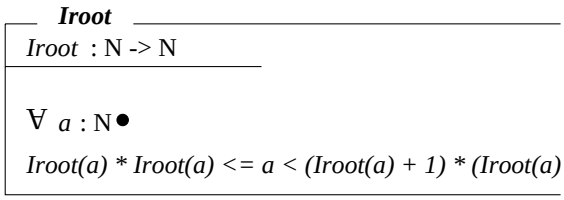
```
function POWERA(X:INTEGER);
begin
 if X = 0 then
    POWERA := 1
 else
    POWERA := 3 * POWERA(X - 1)
end;
```

In any implementation, there will necessarily be an input value, a means of computing three raised to the power of a specified value and a means of outputting the result. However, the form this takes in the majority of cases, and the fact that the implementation can be coded in so many different ways, makes it impractical to consider any automated comparison between specification and implementation. In the definition of our two types of state, we must therefore be careful to ensure that the specification and implementation are comparable in some form. The importance of ensuring a mapping between specification and implementation is exemplified further by the square root function of a natural number.

Example 3: square root function

As a more complicated example, consider the Z specification to calculate the square root of a natural number [Jac97]. Here, the declaration part of the schema tells us that the function *iroot* takes a natural number, and returns a natural number. The declaration of both these values as such numbers excludes the possibility of a negative integer being accepted as input. In the implementation, the responsibility is on the coder to ensure that this does not occur.

The predicate part of the schema returns the *largest* positive natural number iroot whose square is no more than a. For example, an input value of 4 will return the value 2; as will



input values of 5, 6, 7 and 8. An input value of 9 will return the value 3, so too will 10, 11, 12, 13, 14 and 15. Using the fact that the difference of two successive squares, n^2 and $(n+1)^2$, is $2n+1$, the following is the C code corresponding to the specification:

```
int iroot(int a)
{
 int i,term,sum;
 term = 1; sum = 1;
 for (i = 0; sum <= a; i++)
   }
     term = term + 2;
     sum = sum + term;
   }
  return i;
}
```

We can see that the code bears no resemblance to the specification. Clearly, for programs of a certain category, particularly if they are specified in a language such as Z, the three problems outlined previously are significant ones. We therefore accept in this thesis that it would be impractical and unrealistic to develop a model which compared, for example, a Z specification with an arbitrary coded implementation. The notation we use has to provide a mapping between specification and implementation to allow a comparison between the two. In the following section, that notation, allowing the definition and comparison of a non-divergent and partially divergent state, is given. In particular, it will allow a specification and an implementation, expressed in CSP, to be compared.

4.4 Divergence

Given a process, P, let I_P denote the set of identifiers in P. We will denote a specification of P by Π and let $\pi_1, ..., \pi_k$ denote implementations of Π. Given an identifier, i, we define the *state*, s, of a process, P, to be the set

$$\{ \langle i,b \rangle : i \in I_P \}$$

where b is a boolean constant, *true* or *false*; b is defined to be *true* if there is a corresponding identifier in Π with the same current value of i and *false* otherwise. We refer to $\langle i,b \rangle$ as an *s-pair* (an abbreviation of state pair).

A state, $\{s = \langle i,true \rangle : i \in I_P\}$, is said to be *non-divergent*. Otherwise, s is *partially divergent*. A *trace* is a sequence of states, $s_0, s_1 ... s_m$. A trace results from a process starting in an initial state, s_0, and state transitions resulting from *actions*. (We note that this definition of a trace departs from the CSP definition of Hoare [Hoa85], where a trace is defined in terms of sequences of events.) We write $s \xrightarrow{a} s'$ to denote the transition from state s to s' with action a. Clearly, we can represent this situation using a state transition diagram. We will also write $s \xrightarrow{*} s'$ to denote that state s' is *reachable* from s via some (finite) sequence of actions.

Let t be the termination state of a trace (if such a state exists). A trace of a process has *harmless* partial divergence if the trace enters a partially divergent state and t is a non-divergent state. A trace has *harmful partial divergence* if t is partially divergent. A trace is *non-divergent* if every state in the trace is non-divergent.

A trace is *divergent* if it enters a non-terminating state sequence. A process is *divergent* if it has a divergent trace.

In contrast to a trace which exhibits a harmless or harmful partial divergence, we say that a finite trace of an arbitrary process is *totally* non-divergent if every state of the trace is non-divergent. We note that a specification contains only traces comprising non-divergent states; thus, every trace of the specification is totally non-divergent. Some further termination properties are examined in the following section.

4.4.1 Termination properties

In CSP [Hoa85], the $\surd$ event represents successful termination of a process. In [AG94a] the description of processes includes the possibility that a process may terminate after executing a limited number of actions as opposed to being indefinitely invoked[3]. Leaving aside, for the moment, processes which are designed to be non-terminating, we introduce a modification of the $\surd$ event which will prove useful in comparing the end state of processes. We redefine the $\surd$ event to become representative of the *status* of a process if it terminates, so making a distinction between a process terminating in a non-divergent state and one terminating in a partially divergent state. In what follows, and for the purposes of our definition, we take s_I to be the terminating state of the implementation. Remembering our definition of a trace as a sequence of states (and associated actions) we can now state the following:

$\surd_{exp}$ is the end-state status of an implementation trace if s_I is non-divergent.

$\surd_{unexp}$ is the end-state status of an implementation trace if s_I is partially divergent.

At s_I at least one s-pair is of the form $< i_I, false >$.

Having defined these two notions of termination, we can move on to consider some features of transition sequences. By considering sequences of transitions, we can gain some insight into

[3]We consider such processes when later we describe the FDR system in more detail.

the behaviour of an implementation in the context of its traces and termination properties. In the next section, we make a comparison of harmless and harmful partial divergence with that of *forward* and *backward* recovery.

4.4.2 Forward and backward recovery

A harmless partial divergence represents behaviour in which a process recovers from an unspecified state. This recovery can be deliberate or accidental. A close link exists between this concept and recovery in fault-tolerant systems. In the context of a fault-tolerant system, a process can recover from a fault through either: *forward* or *backward* recovery [LA90]. The principle of each is to return a process to a state in which manifestation of that fault is avoided.

Forward error recovery refers to the manipulation of the current system state to prevent a failure from occurring. The system is then capable of making unhindered progress. Failures are anticipated, and dealt with as and when they occur. For example, an exception handling facility for dealing with specific faults in a program, e.g., arithmetic overflow.

Backward error recovery is usually achieved through *recovery points* (often referred to as *checkpoints*) in which the state of a system is restored to a prior, failure-free state. Recovery data (stored at each recovery point) allows the state of the system to be restored. An alternative to recovery points is via a historical record of activity of the system known as an *audit trail* (or log). Finally, we note that both types of recovery described can be applied in a sequential *and* concurrent process environment.

4.5 Inter-dependence of actions and states

As an example of the inter-dependence of states and actions, consider again the case of the sweet vending machine (see Section 3.3). Suppose, in response to a single coin being deposited, the vending machine produced two sweets when it should only have produced one. We can view this sequence of events as a specified first action, yielding a non-divergent state, followed by the second action (dispensing the sweets) which can be seen as an unspecified action yielding a partially divergent state. The behaviour we *expected* of the vending machine was for a single action to produce a single sweet yielding a non-divergent end-state. The transition from a non-divergent to a partially divergent state is therefore made at the point when the second sweet is dispensed. The vending machine has clearly engaged in unspecified behaviour.

We can also show, with reference to the elements of a state transition (s,a,s'), how the type of a destination state s' is determined by the action a. For example, if a sweet costs 10 pence, and 8 pence has already been deposited in the vending machine (this fact would be reflected in the state of the vending machine), then depositing a further 2 pence will cause a sweet to be dispensed (again, reflected in the state of the vending machine). The choice of which coin a customer may deposit next, to make up the cost of a sweet, is determined by the value of coins deposited so far.

In the refusals model of Brookes et al. [BHR84], a non-deterministic choice from a set of possible actions gives rise to a *refusal set* containing the actions refused in preference to the one that is chosen. For example, assume the customer has deposited a 5 pence coin so far. The customer then deposits a further 5 pence coin. This means that the choice of a 2 pence or 1 pence coin has been refused (by the customer) at that point. A customer therefore has choices at each stage of depositing the coins making up the 10 pence. In the refusals model,

the set of traces a process is capable of engaging in represents the combination of choices of action it has at each execution step. In the above example, this is just saying that there are many combinations of coins capable of making up the 10 pence.

4.5.1 Interpretation of an action

First, we view an action as causing some change in the information known about the current state. For example, we can view the state of a vending machine as the cumulative value of coins deposited in the machine (before dispensing a sweet), and the action of depositing further coins as adding more information about the progress of the transaction as a whole.

Consider the following example of a poorly-coded implementation, in which the process repeats the request for an `x` on the channel `left` (hence introducing a redundant action). Assume the specification is to accept the value for `x` just once.

Example 4: a redundant action

```
x = 1

PING = left ? x -> left ? x
```

The second `left` action does not change the information we know about the process at that state. In fact, in our model so far, both states of the implementation just described we would label non-divergent, since `x` holds the correct value at each state, and the process ends with $\surd_{exp}$. We could interchange the two actions with no change in the subsequent end-state type, whether it be $\surd_{exp}$ or $\surd_{unexp}$. In our model of divergence, a state is defined by the set of identifiers and the values they currently hold[4]. We therefore assume the absence of a

[4]In our programmed implementation, a directed graph structure is used to represent both specification and implementation. This provides the necessary information on sequencing.

program-counter (pc), and accept that, in some instances, an action will cause no change in the information known about a state (i.e., the states in a trace are not necessarily distinct). However, particularly in the case of the execution of concurrent processes, we *do* want to emphasise the importance of action *sequencing* in our model, since this sequence determines the types of action and state in a process. As the thesis develops, we show how we avoid the need for a pc, yet still obtain the sequencing information we need. We now consider some more examples to illustrate the importance of the sequence in which actions are executed.

Example 5: a process to increment an identifier

In a CSP setting, consider the following implementation which accepts an identifier `x` on the left channel and then increments it by 1. Consider the following implementation for a specification to increment a value by one.

```
x = 0

IMP = left ? x -> right ! x + 1                                    (1)
```

The cumulative effect of that increment is incorporated into the annotated state information (see Example 1). All states of this implementation are non-divergent, and the process ends $\surd_{exp}$, i.e., the states of the specification and the implementation are equivalent upon termination. We note that the correct sequence of the two actions — *initialise* and then *increment* — is crucial to this program performing its specification function. Consider now the types of state of an implementation which outputs a value of one on the right channel and waits for the value zero on the left channel before terminating, i.e.,

```
x = 0

IMP = right ! x + 1 -> left ? x                                    (2)
```

In the implementation, a value one would be output on the `right` channel, the `left` channel would be left hanging, waiting with a value zero to be accepted. The same *number* of actions are used as in `(1)`, and each action is encountered in the specification. However, all states in `(2)` except the start state would be partially divergent and hence this would be an example of a harmful partial divergence (as we defined it in Section 4.4). The two examples, i.e., `(1)` and `(2)`, illustrate how, by analysing notions of both action and state, and the inter-dependence of the two, we can begin to understand the characteristics of actions as well as those of states. If we wanted to analyse the level of *divergence* of an implementation from its specification in terms of state information (and state information is dependent on the actions which a process executes), then we should include an analysis of the actions which cause those states.

4.5.2 Observable and unobservable actions

Consider the sweet vending machine example again, capable of dispensing two types of sweet. The racks of the vending machine and the selection of the sweets from those racks are assumed to be unobservable (hidden), i.e., cannot be seen by the customer. Another unobservable action would be the acceptance of coins. An observable (non-hidden) action would be the offering of the sweet to the customer, thereby ending that particular transaction. We could view the vending machine as a black-box, into which money is deposited, and sweets are produced.

We have already seen some examples of the way in which the execution of an action can influence the behaviour of a process, irrespective of whether the action is part of a physical real-world object such as a vending machine, or in a non-physical form such as that of a computer program. If we want to compare the actions in a specification with those in an

implementation, then we should choose an alphabet of actions which lends itself readily to this comparison and allows us to enumerate the actions in that process.

4.5.3 The alphabet of a process

Each entity, whether an object in the real world or an abstract process, has a set of pre-defined, valid actions in which it can engage. Milner [Mil89] defines the alphabet *Act* of a process to be the set of labels of a transition system, say $\mathcal{L}$, plus the handshake action τ (the tau action was described briefly in Section 2.4.4). In a communicating process environment, the τ action represents the action of exchanging values on two channels. It is an instantaneous action, as opposed to the synchronised actions we have described so far. In the context of the vending machine, an example of the handshake action τ would represent the offering of the sweets by the machine followed by the customer removing the sweets from the dispensing tray of the vending machine.

Thus far we have described how, as in the case of a vending machine, different states are yielded, and we have touched briefly on the set of specified actions which bring about a desired end-state. Consider, for a moment, the unspecified actions of a vending machine. We are unable to explicitly list these actions, since the alphabet of a vending machine specification contains only specified actions, and its implementation (the physical machine) cannot cater for every eventuality. We cannot really predict the invalid behaviour that machine is likely to encounter in its lifetime.

In the case of a vending machine we can think of a large number of ways a machine may malfunction. However, in the case of a computer program, we can assume a large (but limited) number of ways that the program can go wrong. We therefore assume that the actions causing non-divergent and partially divergent states are finite in number, thus simplifying

our definition of specified and unspecified actions. Consider the following example.

Example 6: alphabet contents

Using CSP, we declare a process X as having channels `left` and `right`, and declare that the only values capable of being passed to those channels are the values `0` and `1`. This gives the following alphabet for process X, comprising eight actions:

```
left ? 0, left ? 1, left ! 0, left ! 1, right ? 0,
right ? 1, right ! 0, right ! 1
```

We note that process X can be either specification or implementation. We further note that in the examples given in this chapter, we use CSP containing only a minimal number of control statements. By using this small set of actions, in which values assigned to identifiers are explicitly stated, it becomes relatively easy to determine statically the actions in the alphabets of a specification and its implementation. The reason for this simplification is simple. In our programmed implementation of the model we have described in this and the preceding chapter, we use CSP whereby for our analysis, we *must* be able to statically determine the specification and implementation alphabets.

4.5.4 Specified and unspecified actions

The previous section showed the need for analysis of actions as well as states of a process, and hinted at a formal definition of a specified and unspecified action. Based on our analysis of the examples given, as a first attempt at a definition of a specified and unspecified action, we could say that:

- a specified action is any action a in the alphabet of an implementation which has an equivalent action in the alphabet of its specification.

- an unspecified action is any action a in the alphabet of an implementation which does not have an equivalent action in the alphabet of its specification.

When we say that an action in the alphabet of an implementation is also in the alphabet of its specification, we are referring to the contents of any identifiers contained in that action, not just their syntactic equivalence. For example, the action `z := x * y` must contain the same values for `x` and `y` in both alphabets. So, `z:= 5 * 4` is not equivalent to `z := 10 * 2` even though both cause the same value for `z` to be yielded.

We accept that it is unlikely that all actions in an implementation will have corresponding syntactic actions in its specification. In the presence of refinement however, *some* actions in the implementation may well be equivalent, in the sense described above, to corresponding actions in the specification. We note in passing that, in the case where the alphabets of the specification and its implementation are the same, i.e., no unspecified actions exist in the implementation, their behaviour need not necessarily be equivalent. We could easily re-order the sequence (and number) of the implementation actions (drawn from that common process alphabet) in such a way that its behaviour differs from that of the specification. In the following section, we formalise three comparison techniques which will be useful in later chapters as part of the empirical investigation and the use of refinement metrics, namely, *enumeration* of the alphabet of a process, *information hiding* and *relabelling*.

4.6 Comparison techniques

Given a process, P, the set of possible actions of P, denoted A_P, is called the *alphabet* of P. Process alphabets clearly provide a means of comparing two processes, as illustrated in Section 4.5.3.

When an implementation refines a specification, we often internalise some set of actions. This process is known as *information hiding*. Given two processes, P and Q, we say that Q is externally equivalent to P if there is a subset $B \subseteq A_Q$ which is hidden and $A_P = A_Q \setminus B$. Informally, we write $P = Q \setminus B$ to denote that P is externally equivalent to Q given that the set of actions B is hidden in Q.

Given two processes, P and Q, we say ϕ is a *relabelling* (function or facility) from P to Q if for all identifiers in P there is a corresponding identifier in Q. If p_i is an identifier in P, and the corresponding identifier in Q is q_i, $i \in \{1,2,...,n\}$, we usually write ϕ in the form $p_1/q_1, ...,p_n/q_n$. We now give some examples to illustrate these three comparison techniques.

Examples of the three techniques

Suppose we have two processes, `P` and `Q`, which are defined as follows:

```
P = left ? x -> right ! x

Q = in ? x -> mid ? x -> mid ! x -> out ! x
```

Then, on assuming the identifier `x` can take the values `0` or `1`, A_P is given by

```
{left ? 0, left ? 1, right ! 0, right ! 1}                    (3)
```

and A_Q is given by

```
{left ? 0, left ? 1, mid ? 0, mid ? 1, mid ! 0,
 mid ! 1, right ! 0, right ! 1}                               (4)
```

The above, i.e., `(3)` and `(4)` yield the enumeration of A_P and A_Q, respectively. Moreover, if we assume that

```
{mid ? x, mid ! x}
```

is hidden, then

```
P =  Q \ {mid ? x, mid ! x}.
```

Finally, `left/in`, `right/out` is a relabelling function from `P` to `Q`.

4.7 Summary

In this chapter we have identified two divergent types of state and the transitions possible between them. A non-divergent type of state can be viewed as specified behaviour. A partially divergent state can be viewed as unspecified behaviour. From this, we were able to determine the possible state transitions between these two types of state and how termination properties were related to the sequence of states of these two types during the execution of a process. We also looked at the effect of actions on the states of a process emphasising the importance of actions to our model. From the previous discussion, it is clear that any model which attempts to compare a specification with its implementation would have to be at a level in which the notation of each could be compared. Using CSP whereby processes are modelled in terms of communication between channels and the techniques outlined earlier, this task becomes practically feasible.

Chapter 5

Refinement metrics

5.1 Introduction

A central objective of this thesis is to be able to analyse the level of divergence between a specification and its implementation in terms of refinement of that specification. In this chapter, we approach this objective by describing metrics that can be applied to both a specification and its implementation.

The use of metrics as a vehicle for comparison of a specification and its implementation is not a new concept. A set of metrics was proposed by Samson et al. [SDN+89] to compare algebraic specification languages and their implementation languages (see Chapter 2). Metrics for measuring attributes of the Z specification language have also been proposed and investigated [BWW91]. The major difference between this body of work and that described herein is that the latter uses CSP as the medium for expressing **both** the specification and its implementation; moreover, we use the same metrics for specification and its implementation to monitor refinement. As far as we know, previous work in this area has used different languages to express a specification and its implementation and correspondingly different

metrics to monitor refinement.

Since the emphasis in the thesis has been on a characterisation of the types of state in a process (and to a lesser extent the actions of a process), the refinement metrics will be predominantly cast in terms of the states and actions of the process; the definitions of our two types of state lend themselves easily to this type of analysis as they allow the comparison to be made at a very fine level of granularity.

The set of metrics proposed hereafter are not intended to be a definitive set. The metrics were chosen for the purpose of highlighting differences between specification and implementation when both are expressed in CSP. It was not evident during the development of the metrics which metrics would prove most useful. Indeed, it transpired after the initial set of experiments that some of the proposed metrics were only applicable to Type I CSP systems (i.e., bit-protocol problems); further metrics had to be developed tailored to Type II CSP systems (i.e., those involving a high degree of recursion, such as the Towers of Hanoi). A key feature of the proposed metrics therefore is that they encompass as far as possible the features of CSP which relate to the model defined in Chapters 3 and 4.

The tailoring of metrics to suit the application domain emphasises an important lesson learnt from the work in this thesis. Even when using the same notation (in this case CSP), there is no generally applicable set of metrics; there has to be a certain amount of reflection and flexibility in the choice and interpretation of metrics. However, in the first instance, obvious metrics to investigate were those based on the features of a typical CSP process and that is the basic reason for the choice of the twelve metrics. In addition, as part of the overall research approach, the seeding of faults into the implementation during the refinement process and the subsequent effect that this had on the metrics values was also investigated (although, it must be stated, to a lesser degree than the main focus of the work which was

the examination of the refinement process in the absence of faults).

In the next section, we describe the refinement metrics developed, outlining why each is useful as a refinement metric and how a developer may benefit from the information it provides. In Section 5.3 we describe the motivational issues regarding the development of our metrics and in Section 5.4 we discuss their empirical and theoretical aspects. Finally, in Section 5.5 we present our conclusions.

5.2 The metrics developed

The metrics developed embody the principle of information hiding in the form of processes whose internal behaviour is hidden from the outside world; the process is viewed as a black-box. As a result of refinement, this hidden behaviour can take the form of sub-processes all of whose behaviour may be hidden by the developer. Henceforward, we define a refinement step as:

> *a point in the development of an implementation at which a comparison of the specification and its implementation is made; this comparison determines whether the implementation refines the specification.*

A refinement step can be viewed as a checkpoint of a process (following the principle of step-wise refinement). Given a top-level description (specification) of a process, successive refinements are made until the developer decides that the implementation is at a low enough level of granularity and satisfies the original specification. We then talk in terms of "successive refinement steps" to produce an implementation from a specification. This principle is embodied in our refinement metrics. The list of the twelve refinement metrics proposed is given in Table 5.1. One of the main aims of the empirical evaluation undertaken was to

Metric	Description
1	Number of states in the specification
2	Number of states in the implementation
3	Number of distinct actions in the specification
4	Number of distinct actions in the implementation
5	Number of hidden actions in the implementation
6	Number of visible actions in the implementation
7	Number of sub-processes in the specification
8	Number of sub-processes in the implementation
9	Number of additional actions in the implementation
10	Number of non-divergent states in the implementation
11	Number of transitions in the implementation
12	Number of harmless partially divergent states

Table 5.1: Refinement metrics

examine the changes in selected metrics in those cases where the cardinality of the state space of both the specification and its implementation was systematically increased.

The state space of a CSP process can be increased in a controlled way by increasing the number of values which can be passed down the channels defined in that process; such an increase can be achieved by simply editing the CSP code to increase the data values the process is able to use. The exact effects are analysed in Chapters 6 and 7.

In the sequel, we examine each of these metrics in turn, providing a justification for each, considering any limitations in their use and commenting on how the value of the metric may change as the size of the state space increases.

1. Number of states in the specification

Every state in the specification is unique; each state identifies the current status of the process in terms of the progress it has made. State information relating to the specification is available when a refinement check is run between the specification and its implementation (using the facilities of the FDR model-checker). For the two examples used in the following

chapter, the specification is assumed to remain unchanged during refinement [1]. The number of states in the specification will increase if the number of values declared in the specification, capable of being passed down a channel, is increased. A specification is normally composed of a (small) set of channels and other operators and, hence, we would expect the number of states in the specification to expand relatively slowly vis-*à*-vis the corresponding number in the implementation.

2. Number of states in the implementation

This metric is produced when a refinement check is run using the FDR model-checker. If the implementation refines the specification then, by the definition of refinement, there should be more states generated by the implementation than by the specification. The number of additional states in the implementation would indicate the extent to which the developer has refined the implementation in terms of adding extra behaviour. For example, adding more channels increases the capabilities of an implementation by increasing the opportunities for communication. We would expect the gap (in terms of the size of the respective state spaces) between specification and implementation to grow at an increasing rate due to refinement.

The number of states in the implementation will increase if the number of values capable of being passed down a channel is increased; since the implementation will normally comprise a larger number of channels than the specification, the number of states in an implementation will tend to grow at a much faster rate than that of the specification when extra values are added to a channel's capability. A study of how the state space increases as data values increase is examined in more detail in the subsequent two chapters. For different application

[1] One interesting extension to the work contained in this thesis might be to analyse the effects in terms of the metrics values of modifying the specification, i.e., removing the assumption that the specification models the requirements exactly. A small sample of tests were made under these conditions, the results of which are described in the final chapter.

domains, the rate of increase of the state space of the implementation is likely to vary enormously. For example, it was found that increasing the state space of Type II systems was far more problematic than for Type I systems.

3. Number of distinct actions in the specification

The specification will contain a minimum set of actions which accurately capture the stated requirements. From this minimum set, the implementation will be derived. In this thesis, *distinct* actions refers to the combination of possible values which a channel is capable of holding. In the programmed implementation of our model, an identifier is expressed in terms of a CSP *channel* capable, at any one point, of holding more than one value. For example, for a channel

```
left ? x
```

if `x` can take the values zero or one, then this represents two distinct actions. The number of distinct actions can be increased by increasing the number of values in the `datatype` set within the body of the CSP process. Increasing the number of values capable of being passed down a channel will therefore increase the number of distinct actions in the specification. We note, in passing, that the number of distinct actions (in both the specification and its implementation) can obviously be increased by the addition of new channels as well as by increasing the capacity of existing channels. We also note that, in a sense, before refinement starts, the implementation and specification could be considered as identical (i.e., they are one and the same process).

4. Number of distinct actions in the implementation

It may be useful for a developer to know the number of distinct actions in the implementation in order to quantify the additional capabilities of either introducing new channels as part of the refinement process or enhancing existing channels. Extra behaviour can be either hidden *or* observable. For example, the addition of the `mid` channel used in earlier examples allowed a single-place buffer to become a two-place buffer; the extra capability was hidden. Increasing the number of values capable of being passed down a channel will increase the number of distinct actions in the implementation.

5. Number of hidden actions in the implementation

The number of hidden actions in the implementation indicates the extent of information hiding invested in the CSP process by the developer. What we are trying to capture with this metric is the number of actions which the process is capable of executing internally (unobservably). This metric may give an indication of the extent to which the developer has thought about the design of the process, since the addition of hidden behaviour to the implementation indicates that the developer is aware of specification behaviour. In other words, the developer is fully aware of the possibilities for adding hidden behaviour to the implementation through extensive knowledge of the behaviour of the specification.

Just as in the object-oriented paradigm, where decisions regarding encapsulation and coupling need to be thought through carefully, so the same is true of CSP-based processes. The extent of hiding behaviour in a CSP process may reflect a well-thought out design by its conformance to the black-box approach.

6. Number of visible actions in the implementation

A developer may be interested to know the number of actions which can be executed observably; in many cases, those actions which can be observed visibly in the implementation will have corresponding actions in the specification (since, strictly speaking, all refined behaviour should be hidden). This metric may help to identify commonalities between the implementation and specification; in the context of a fault in the implementation, knowledge of which parts of the implementation overlap with the specification may help in the fault detection process. In Chapter 7, limited fault-based analysis is described for each of the four systems investigated [2]. We note that, although this metric is derivable by subtracting the number of hidden actions from the number of distinct actions in the implementation, the distinction may serve a useful purpose in the ensuing chapter when analysing scatter plots and correlation values.

7. Number of sub-processes in the specification

The specification is unlikely to contain the same level of decomposition as the implementation and therefore is unlikely to contain the same number of sub-processes. Often, a specification will contain zero sub-processes, indicating that all of its behaviour is observable. One such example is the single-place buffer, used as an example in Chapters 1, 3 and 4, which contained zero sub-processes.

[2] Available literature in the area of fault seeding is large. The analysis of fault seeding in this thesis was only preliminary in the sense that the conclusions are tentative, and hence need to be examined more thoroughly. Enough data was collected, however, to at least get an idea of the effect of seeding faults within CSP processes.

8. Number of sub-processes in the implementation

An important part of incremental development is the ability to construct a larger process by *glueing* together smaller, stand-alone processes. In [AG94b], it was shown how the architecture of a system could be viewed in terms of components and the glue holding those components together.

As developers we would be interested to know the number of sub-processes belonging to a process in order to measure the extent of decomposition that we had chosen to incorporate into the implementation. In terms of program maintenance, where the maintainer is often not the original coder, knowing the extent of decomposition might be useful in pinpointing the source of required maintenance, whether corrective, preventative or perfective. We would also be interested in the number of states attributable to each sub-process, if, for example, we wanted to compare two sub-processes for efficiency, or to determine where the majority of the process behaviour had been invested. This metric is produced by running a refinement check of a specification against the corresponding implementation.

9. Number of additional actions in the implementation

This is the number of actions in the implementation not present in the specification. It gives an indication of the extra behaviour that the implementation is capable of engaging in. As an implementation is refined, this metric should increase.

We note that preliminary empirical evaluation revealed that the metric counting the number of additional actions in the specification is inappropriate. The implementation should contain at least the same number of actions as the specification (it should not normally contain fewer actions).

10. Number of non-divergent states in the implementation

This metric indicates the number of states in the implementation which, after relabelling and hiding, have corresponding states in the specification. To obtain this metric requires an analysis of the set of states in the implementation. An implementation which refines a specification will contain only non-divergent states. If there are faults in the implementation, not all states will be non-divergent (i.e., they will be classed as partially divergent states). Equally, if a state in the implementation does not appear in the specification, it will be classed as a partially divergent state.

11. Number of transitions in the implementation

For problems which are, by their nature, communication-based, a large number of the underlying state transitions will be handshake (or tau) operations representing the exchange of data between the different sub-processes of the process. This metric would therefore be a measure of the extent of communication-based computation in a process; in theory, different applications within the same domain should exhibit similar levels of communication. For example, we would expect the multiplexed buffer and alternating bit protocol examples to exhibit broadly similar communication patterns.

On the other hand, domains such as those typified by the Dining Philosophers and Towers of Hanoi problems may exhibit completely different communication patterns. Investigation of this sort of feature is central to the analyses contained in the next two chapters.

12. Number of harmless partially divergent states

As well as the number of transitions in the implementation, the number of harmless partially divergent states was also collected. Whilst a strong relationship between this metric and

the previous metric was expected for implementations which refine their specifications, the relationship between the two metrics in other circumstances such as part-refinement (i.e., checking just part of the implementation against the specification) is not so clear. In Chapters 6 and 7 we investigate the relationship between these two metrics.

We next discuss some of the issues raised by the proposed metrics.

5.2.1 Discussion

Although there has been considerable interest in the use of metrics for various types and sizes of system, see [CS00] for example, to date few metrics have been developed to capture features of CSP refinement. Refinement has been viewed in the past as a technique of program development in the formal methods community, rather than as an applicable discipline in the empirical software engineering domain.

More importantly, very few sets of proposed metrics capture either features of the design or features of the development process. Capturing metrics of the process rather than the product has obvious implications for the overall time and effort of development. If we can understand problems associated with the development process, we will be in a better position to understand some of the problems associated with the development life cycle such as late delivery of projects or over-budget projects.

We note firstly that the refinement metrics proposed in this thesis are all capturable automatically by a programmed implementation. In any system containing a large number of states, manual collection would be error-prone, cumbersome and time-consuming. We note secondly that the refinement metrics proposed and our model of divergence were designed to be applied to a subset of processes, i.e., to those expressed in CSP. They may not necessarily be appropriate in other environments, for example, processes expressed in high-level lan-

guages such as C++ [Str94]. The metrics chosen are all counting metrics, and are relatively simple to collect and interpret. We see this as a positive feature, since simple metrics are often the most effective. We note finally that:

- since all states in the specification are non-divergent, the number of non-divergent states in the specification is the same as its number of states;
- the number of partially divergent states in an implementation can be calculated by subtracting the number of non-divergent states from the total number of states.

From these proposed metrics, divergence can then be expressed at varying levels of granularity, in terms of any of the following:

1. The number of extra states present in the implementation.
2. The number of extra actions present in the implementation at each refinement step. This would indicate the extent to which extra computation had been introduced into the implementation at that refinement step.
3. The type of each state and action within an implementation.
4. For each sub-process within an implementation, the states, actions and the type of each state. At different levels of abstraction, we can also determine the level of information hiding in the implementation in terms of sub-processes and hidden actions.

5.2.2 Normalisation of the metrics values

One consideration which should not be overlooked is the possibility of normalising our refinement metrics to take account of the difference in size between one process and another (i.e., the specification and its implementation). However, the notation of CSP processes is unlike

that of programs written in the procedural or object-oriented paradigm. This makes it problematic to normalise on any attributes of a CSP process, such as the number of lines of code. The counting metrics developed allow meaningful comparisons to be made between processes based on states, actions, extent of hiding, etc. We expect, in general, the implementation to be *larger* than the specification (in terms of actions and states) and this would be reflected in the metrics values obtained from comparison of a specification and its implementation. The same principle applies to the comparison of one implementation with another, in the case where we would like to know which contains the greater number of states, actions and extent of hiding, for example.

As an example of the CSP processes we will be describing in the following chapter (and to illustrate some of the features relevant to the metrics just described and the analysis in the next chapter), consider the same example as that given in Chapter 1, of the single-place buffer implemented by using two sub-processes.

5.2.3 Example

The specification is represented by `COPY` and the implementation by `SYSTEM`.

```
DATATYPE = {0,1}
channel left,right : DATATYPE
channel mid : DATATYPE

COPY = left ? x -> right ! x -> COPY

SEND = left ? x -> mid ! x -> SEND
REC  = mid ? x -> right ! x -> REC

SYSTEM = (SEND [| {| mid |} |] REC) \ {| mid |}
```

In Chapter 4 we described how, through the use of both hidden identifiers (or channels as they are more commonly known) and relabelling, an implementation can be observationally

equivalent to its specification, but may also contain extra identifiers which may or may not be hidden. Our definitions of a non-divergent and partially divergent state were based on the equivalence or otherwise of states after hiding and relabelling had been taken into consideration. The example just given encapsulates all of these CSP features.

We can now view the distinction between a non-divergent state and a partially divergent state as necessary for the development of our refinement metrics, not least because they allow questions such as the following to be answered:

1. For how long (in terms of the number of state transitions) is the implementation in a partially divergent state? This would tell us for how long a process is exhibiting unspecified behaviour.

2. Which sub-sequences of states in the implementation are non-divergent states?

If we know that the implementation makes a transition from a non-divergent state to another non-divergent state, then both states contain no other channels other than those that were either hidden or included in the specification; the implementation refines the specification. Equally, if an implementation makes a transition from a partially divergent state to a partially divergent state then both states contain extra channels which do not appear in the specification. These are all conclusions we can draw from our definitions of non-divergent and partially divergent state.

In the following section, we describe the empirical aspects of our approach.

5.3 Development of the metrics

Empirical evaluation can be used to investigate the association between proposed software metrics and other indicators of software quality such as maintainability or understandabil-

ity. What we are attempting to do by validating metrics is to show how they can help us understand how software is constructed. Validation in this sense can take a number of forms including hypothesis testing, cost models and fault injection techniques.

The examples we present in the following chapter represent an empirical investigation of our metrics in order to establish their usefulness to a developer, based on our model of divergence. On its own, the empirical evaluation of metrics is not sufficient; the theoretical aspects of metrics must also be considered.

In the next section, issues relating to the theoretical nature of our metrics are discussed.

5.3.1 Criteria for theoretical validation

In measuring attributes of software, Fenton [FP96] poses several questions, applicable for real-world entities, but more difficult to answer when considering software. The questions posed highlight the more abstract nature of software and the problems associated with software measurement. For each of the following points, we consider the role of **our** model of divergence and how our refinement metrics figure in terms of the theoretical issues raised:

1. How much must we know about a software attribute before it is reasonable to consider measuring it? Our model makes the distinction between two types of state, for example. This then allows metrics to be produced based on an underlying model of these two features.

2. How do we know if we have really measured the software attribute we wanted to measure? Since refinement is made to an implementation in terms of extra behaviour (and extra behaviour can be expressed in terms of states and actions), then it is fair to assume that the metrics proposed do capture aspects of refinement.

3. What meaningful statements can we make about a software attribute and the entities that possess it? Since our metrics are counting metrics, we can easily compare processes in terms of numbers of states, actions, etc., and make valid judgements on the differences between any finite number of processes in these terms.

4. What meaningful operations can we perform on measures? Mathematical operations are meaningful on counting metrics (based on the absolute scale of measurement). To say that one process has twice the number of states, actions or sub-processes compared to another process has an intuitive and mathematical meaning.

The theoretical approach to the validation of metrics requires us to clarify what attributes of software we are measuring, and how we go about measuring those attributes [Fen94, KPF95, BBM96]; a metric must measure what it claims to measure.

We next introduce the idea of the representation condition and demonstrate how our metrics conform to this and other theoretical principles of metrics.

Fenton [FP96] describes the *representation condition*, satisfaction of which is the prerequisite for any metric to be viewed as valid. The representation condition states that any measurement mapping must map entities into numbers, and empirical relations into numerical relations, such that those empirical relations are preserved. In other words, our observations in the real world must be reflected in the numerical values we obtain from the mathematical world. Since our measures of refinement are expressed in terms of simple counts, we expect the metrics collected to reflect the observed refinement values. For example, if one process appears to contain a larger amount of information hiding than another process, then this will be reflected in the metrics obtained (in particular, in the number of sub-processes and hidden actions).

Kitchenham et al. describe a list of features of metrics which must hold for any metric to be valid [KPF95]. Metrics are usually based on internal (low-level) attributes, derived from external (high-level) attributes. For our model, the internal directly measurable software attributes (derived from software fidelity) are encapsulated in the twelve proposed refinement metrics.

The following criteria must hold for a direct metric (i.e., a metric associated with a directly measurable software attribute) to be considered valid:

- For a software attribute to be measurable, it must allow different entities to be distinguished from one another. As an example, it is possible for one object to be travelling faster than another object. Analogously in the context of software, it is feasible for one OO class to have more methods than another. From the example expressed in CSP, given earlier in this chapter, it is obvious that one process can have more states than another process. It is expected, for example, that an implementation will, through the process of refinement, contain more states than the corresponding specification.

- A valid metrics measure must obey the representation condition, that is, it must preserve all intuitive notions about the software attribute under consideration and the way in which the measure distinguishes between entities. For example, one object may indeed be travelling faster than another object, and when measured in miles per hour (mph) this observation holds true. Applied to software, counting the number of methods in two OO classes, which appear different in size, reveals that one contains more methods than the other. The same is true in the case of our metrics in terms of the number of distinct actions and sub-processes, for example.

- Each unit of a software attribute contributing to a valid measure is equivalent. Con-

sidering the speed of a particular object, the unit difference between 60 mph and 61 mph is the same as that between 61 mph and 62 mph, and so on. Applied to software, each OO class method is considered to be the same (it adds one to the total number of methods). Since each of our refinement metrics are counting metrics, each unit, whether state or sub-process, for example, is considered to be the same. No distinction is made between one state or another in terms of its unitary value; the same can be said of sub-processes.

- Different entities can have the same software attribute value within the limits of measurement error. Continuing the analogy, one object may indeed be travelling at an identical speed to another object. Applied to software, two OO classes may well have the same number of methods, and in terms of our refinement metrics it is quite feasible for one process to have the same number of states or sub-processes, for example, as another process.

The ability to quantify the extent to which an implementation refines a specification, by using our set of proposed metrics, can be put to use in various ways.

Firstly, we can decide which of several completed implementations refine a specification the most. In such circumstances we may want to choose the implementation which is *furthest* (in terms of divergence) from the specification, as this would represent the most refined implementation. However, the meaning of *furthest* would have to be clarified. One option would be to use the number of states as the distance metric. The difference between two CSP processes would then be expressed in terms of the difference in the number of states. Another option may be to use the degree of hiding in the implementation (given by the number of sub-processes). Some of these issues, including the notion of a refinement ordering, were investigated in [MMM97] and the purpose of the next two chapters is to explore some of

these issues in a CSP setting.

Secondly, if several refinement options are available to choose from, then producing a metric which captures the effect of that refinement (in terms of metrics values) can provide valuable information to the developer in deciding which refinement option to choose. In our case, the metrics values could therefore be used to guide the development process.

5.3.2 Other metrics collected and considered

As well as the metrics described in earlier sections of this chapter, the CPU timings for each refinement run were also collected. The motivation for collecting this data was that timing information may shed light on features of a refinement check which the other metrics may have disguised. This was particularly important in the context of the different application domains being investigated. Different styles in the way refinement was achieved lead to interesting differences in the time it took to run a refinement check for specification and implementation. The two non-protocol based applications investigated (the Dining Philosophers and Towers of Hanoi) showed very different characteristics to those of the first two (multiplexed buffers and the alternating bit protocol) in terms of the way they were written and the size of the state space examined. The nature of CSP processes also meant that metrics based on program complexity [Hal77], or metrics based on lines of code, were less appropriate than they are in other programming languages. The depth of recursion used in each of the problems considered also made counts such as lines of code inappropriate.

As a further note on the empirical evaluation, in the case of implementations with a large state space, each refinement run often required a time interval of one and a half hours between starting and completing. Some refinement runs had to be terminated because virtual memory on the server being used (rhea.dcs.bbk.ac.uk) was exceeded. In such cases,

no metrics could be obtained from the refinement checks. We postpone a full explanation of the empirical evaluation until the next two chapters.

5.4 Conclusion

In this chapter, we have described the development of the refinement metrics we will be using in our comparison of a specification and its implementation. We looked at the empirical and theoretical nature of metrics and a set metrics were then proposed as a means of quantifying divergence in the sense we have described in Chapter 4. These metrics capture elements of the process as an implementation is successively refined and may give an insight into the refinement process and the choices available to a developer during the process of refinement. The metrics may also shed some light on the behaviour of CSP in the presence of faults in the implementation.

In the next two chapters, we present our empirical evaluation. Four examples were chosen as the basis of the empirical evaluation. As well as looking at metrics from the refinement (and non-refinement) point of view for each of these example problems, the effect of introducing selected faults into the implementation was also touched upon. Empirical evaluation will help to establish how useful the metrics are in reflecting features of the refinement process and how effectively they capture properties in different application domains.

Chapter 6

Empirical evaluation (Type I CSP Systems)

6.1 Introduction

In the previous chapter, a set of metrics was proposed which forms the basis of an empirical study of divergence. The aim of developing these metrics is to investigate the features of CSP processes during the process of refinement. In this chapter we describe the first part of this empirical study in which these metrics are collected, detailing the types of process investigated, the data analysis undertaken and an interpretation of the results for the two Type I CSP systems. An assessment of the proposed set of metrics was a main objective of the thesis (see Chapter 1, Section 1.3). A major part of this chapter is to analyse their behaviour in the context of refinement and non-refinement (this was the other objective of the thesis stated in Chapter 1). In other words, when, firstly, an implementation satisfies (refines) the specification and, secondly, when it does not (in the latter case, because it may contain behaviour unrecognised by the specification or lacks behaviour required by the specification,

i.e., non-refinement of the specification). An equally important part of this chapter will be to compare the results from the two Type I CSP systems analysed. The proposed metrics are the means by which differences between the specification and implementation are made explicit.

In Section 6.2, we describe the motivation for the empirical study. Section 6.3 gives a brief description of the two systems investigated, how the study was carried out, and the data analysis for the first two problems in the context of refinement. Interpretations of the results from this analysis are also given, supported by scatter plots and histograms. Section 6.4 gives a similar analysis to that of Section 6.3, but in the context of non-refinement. Finally some conclusions are presented in Section 6.5.

6.2 Motivation

The motivation for the empirical study is:

Firstly, there are likely to be features of the refinement process of CSP systems which have yet to be uncovered through a study of this sort. The software engineering community is still trying to learn about software construction and the behaviour of the programmers that write the software. Indeed, major issues such as the use of inheritance is the subject of debate within the object-oriented paradigm; for example, issues to do with the use of inheritance. In short, empirically-based research is still in its infancy and many more studies in all aspects of software engineering need to be undertaken.

Secondly, very little is understood about the architecture of systems across different application domains and differently sized problems. For example, whether all solutions to bit-protocol problems exhibit similar levels of information hiding or exhibit similar patterns of communication. Equally, whether highly recursive systems (such as those described in

the next chapter) exhibit completely different features to those less recursively-oriented. An important objective of the empirical study is to try and improve our understanding in these areas.

Thirdly, CSP systems lend themselves naturally to decomposition and synthesis. For example, implementation of the two-place buffer (introduced in Chapter 1) comprised two processes communicating via an intermediate channel.

The empirical study will examine features of CSP systems as they are combined with (or decomposed further into) other such systems. This means that in addition to situations where the implementation refines the specification, there will also be an analysis of situations where the implementation does not refine the specification; in such cases, instead of focusing on refinement, we look at the differences between the specification and its implementation as the latter is constructed. This situation would also arise in the case of fault injection where we would like to be able to see what differences there are between a specification and its implementation as the latter is being constructed (a topic we very briefly address in Chapter 7).

Four different CSP systems were chosen for the empirical study. In this chapter, the first two are empirically investigated in the context of refinement and non-refinement. These are two bit-protocol applications; that is, applications whose solution is largely based around communication between sub-processes to form the overall process (Type I CSP systems). In the next chapter, the other two, representing classical problems in computer science, are also empirically investigated (Type II CSP systems).

As well as a different emphasis on the underlying problems they addressed, there was also a difference in the type of computing resource used by each pair of CSP systems; Type I CSP systems used predominantly system main memory, whereas Type II CSP systems

tended to use system stack space due to their dependance on recursive structures. We also note that Type I CSP systems used more interleaving and non-deterministic operators than Type II CSP systems (these are key CSP semantic features).

We note further that the two types of system identified (Type I and Type II) are not exhaustive; this taxonomy is more for convenience and for clarity of explanations in the thesis. The Railway Crossing system, for example, examined briefly in Chapter 7, could well form the basis for another type of CSP system since it exhibits properties somewhere between Type I and Type II systems.

We remark at this point that the four CSP systems were originally presented in [For97]; they have, however, been modified in certain ways to control the refinement checks, and later on, in the context of fault injection, to influence the behaviour of the refinement process.

6.3 The four examples

The four CSP systems used for the empirical study were:

1. **System one**: A Multiplexed Buffer
2. **System two**: An Alternating Bit Protocol
3. **System three**: The Towers of Hanoi
4. **System four**: The Dining Philosophers

The choice of these four systems was made on the basis that they should be of realistic size (in terms of the problem which they address), capable of being expanded to a large state space. By large state space, we expected the upper limit (through prior pilot experiments) to be around four hundred thousand generated states. This value could have been raised,

but would have started to compromise limits on the virtual memory of the machine on which the refinements were being run.

There should be two systems within each of the two domains identified (Type I and Type II) to allow a comparison both between and within domains. The domains themselves should also be widely contrasting in their nature. Hence, there are two systems from the bit-oriented domain, and two from the classical computer science domain. The bit-oriented domain is typical of the types of problem suited to CSP; the last two systems are more problem based and highly recursive in nature. The fact that they were highly recursive highlighted its own problems, not least of which were the large internal structures, for example, directed graphs generated by the FDR model-checker; we describe these issues when we analyse Systems three and four in the next chapter.

In the following sections, the two bit-protocol systems are presented in detail. For each, the experiments undertaken are also described and the results analysed; comparisons between the two systems are presented at various points. For each of the two systems the following tests were carried out:

1. Refinement tests for each of the two systems – this is where the implementation refines the specification.

2. Non-refinement tests for each of the two systems – to understand how systems are composed (synthesised) to construct the implementation and the changing features of systems as this happens; a large number of refinement tests were carried out and metrics were collected for such tests of sub-processes against the specification.

We note that all four systems by their specific nature imposed some limitations on what refinement checks could be carried out. For the first two systems we could only expand the

size of the state space up to a limit imposed by the machine on which they were running. This applied to both the refinement tests and the non-refinement tests. These limitations also explain why, for each of the four CSP systems analysed, different numbers of refinement checks were undertaken.

The CSP code for the four systems is contained in Appendices A to D. However, in the following descriptions, fragments of the CSP code from these appendices are reproduced to aid the reader's comprehension of each system.

6.3.1 System one: a multiplexed buffer system

The full CSP code for the multiplexed buffer system is given in Appendix A. The following description is on the whole from [For97].

> The idea of this example is to multiplex a number of buffers down a pair of channels. They can all be in one direction, or there might be some in both directions. The techniques demonstrated here work for any number of buffers, and any types of transmission. The number of states in the system can be easily increased to any desired size by increasing either the number of buffers, or the size of the transmitted dataset.

Figure 6.1 illustrates the overall structure of the system. The specification of such a system can be expressed simply as

```
Copy(i) = left(i) ? x -> right(i) ! x -> Copy(i)
Spec = ||| i:Tag @ Copy(i)
```

where `i` represents the instance of a transmitter or receiver. The `|||` symbol represents an interleaving of behaviour between transmitters and receivers. The implementation (Figure 6.1)

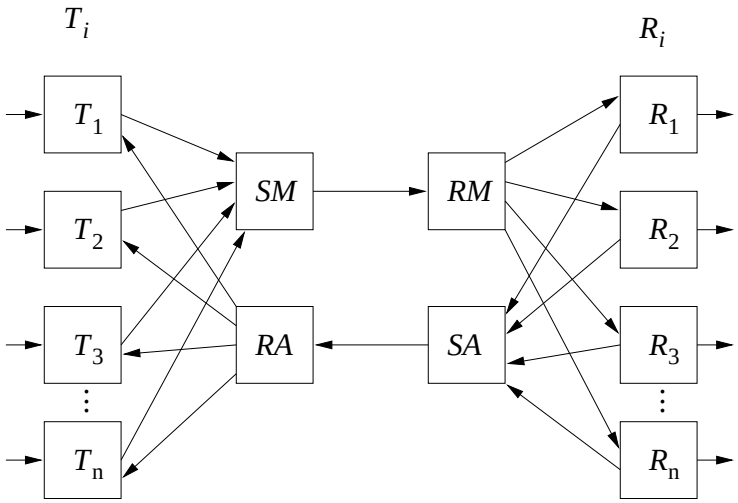


Figure 6.1: Multiplexed buffers with acknowledgment

consists of n transmitters (T_i), the same number of receivers and four other processes which handle communications between the transmitters and receivers. *SM* (Send Message) multiplexes transmitted data and *RM* (Receive Message) demultiplexes messages. *RA* (Receive Acknowledge) and *SA* (Send Acknowledge) handle the acknowledgments for these messages.

Key to the understanding of this and the following problem was the ability to manually increase the state space by altering the values that could be passed down the channels of a process. The following datatype declarations are for `Tag` and `Data`; these were changed to increase the size of the state space, and hence generate a new set of metrics.

```
datatype Tag  = t1 | t2 | t3
datatype Data = d1 | d2
```

These values could be changed as and when necessary, and the refinement check re-run. The overall system (implementation) can be viewed as an `LHS` communicating with an `RHS` with hidden channels `mess` and `ack`, namely,

```
System = (LHS [|{|mess, ack|}|] RHS) \ {|mess,ack|}
```

LHS (see (1) below) is a composition of multiple transmitters (see definition of Txs given by (2)), with SndMess and RcvAck being interleaving operations. The channels

```
snd_mess
rcv_ack
```

are hidden. In this case, LHS is defined by

```
LHS = (Txs [|{|snd_mess, rcv_ack|}|]
           (SndMess ||| RcvAck))\{|snd_mess, rcv_ack|}                    (1)
```

Looking at each of the three sub-processes of LHS in turn, we note that the set of transmitters Tx(i) is defined by

```
Txs = ||| i:Tag @ Tx(i)                                                   (2)
Tx(i) = left.i ? x -> snd_mess.i ! x -> rcv_ack.i -> Tx(i)
```

```
SndMess = [] i:Tag @ (snd_mess.i ? x -> mess ! i.x -> SndMess)
```

```
RcvAck = ack ? i -> rcv_ack.i -> RcvAck
```

Correspondingly, RHS is defined by

```
RHS = (Rxs [|{|rcv_mess, snd_ack|}|]
           (RcvMess ||| SndAck))\{|rcv_mess, snd_ack|}
Rx(i) = rcv_mess.i ? x -> right.i ! x -> snd_ack.i -> Rx(i)
Rxs = ||| i:Tag @ Rx(i)

RcvMess = mess ? i.x -> rcv_mess.i ! x -> RcvMess
SndAck = [] i:Tag @ (snd_ack.i  -> ack ! i -> SndAck)
```

6.3.2 Refinement details

The first experiment for System one involved a set of thirty-seven refinement checks for the following asertion:

```
assert Spec [FD= System
```

The following results were produced by the FDR model-checker with three `Tags` and one `Data` value as parameters (we use this data configuration as an example to illustrate the typical output from the FDR model-checker).

```
Checking mbuff.csp

Starting...
Compiling...
Reading...
Loading... done

Starting...
Compiling...
Reading...
Loading... done
-- Normalising specification
-- Normalisation complete
Starting timer
About to start refinement
Refinement check:
Refine checked 1404 states
With 4056 transitions
Stopped timer
Resource   Start              End                 Elapsed
Wall time   8662953             8662954                   0
CPU (self)        0                   0                   0
CPU (sys)         0                   0                   0
(inc children)
CPU (self)        0                   0                   0
CPU (sys)         0                   0                   0
true
```

The output reflects the fact that the implementation refines the specification (the word `true` at the end of the output indicates that the implementation refines the specification). For each

of the thirty-seven refinement checks, the value of `Tag` or `Data` was changed accordingly. Each refinement check produced a set of the twelve metrics described in Chapter 5. Interestingly, because of the limitation on virtual memory of the machine on which the tests were being run (rhea.dcs.bbk.ac.uk), it became impossible to obtain metrics beyond a system with five `Tags` and three `Data` values, i.e., with the following datatype definitions:

```
datatype Tag  = t1 | t2 | t3 | t4 | t5
datatype Data = d1 | d2 | d3
```

We note that, in some cases, refinement jobs were still hanging two months later inside the system.

6.3.3 Summary data

Summary metrics for System one are presented in Table 6.1. The columns represent the minimum, maximum, median, mean and standard deviation values for all twelve metrics for the thirty-seven refinement checks of `Spec` against `System`. The mean and median values have been rounded up or down where appropriate.

Metric	min.	max.	median	mean	std.dev.
1 Number of states in the specification	4	2401	100	371	542.22
2 Number of states in the implementation	16	364440	10584	37415	70589.11
3 Number of distinct actions in the specification	8	62	26	30	15.78
4 Number of distinct actions in the implementation	20	161	71	80	40.53
5 Number of hidden actions in the implementation	4	33	15	17	8.28
6 Number of visible actions in the implementation	16	128	56	63	32.27
7 Number of sub-processes in the specification	0	10	4	4	2.19
8 Number of sub-processes in the implementation	13	29	21	21	4.37
9 Number of additional actions in the implementation	12	99	45	50	24.83
10 Number of non-divergent states in the implementation	16	364640	10584	37415	107316.00
11 Number of transitions in the implementation	12	676128	17100	70984	152315.50
12 Number of harmless partially divergent states	6	338064	8550	35492	67150.55

Table 6.1: Summary metrics for System one

All metrics values were obtained independently bearing in mind that metric 5 added to metric 6 gives metric 4. We also observe that for implementations which refine the specification (in contrast to non-refinement), the number of non-divergent states in the implementation (metric 10) equals the number of states in the implementation (metric 2).

It is noticeable that the maximum, median and mean number of transitions in the implementation (metric 11) is larger than the corresponding values for the number of states in the implementation (metric 2). This is because the 'transitions metric' includes the tau (or handshake) action [Mil89], which is not included in the digraph from which the metric for the number of states in the implementation is computed. The exception to this trend is the minimum value for metric 11, which is smaller than that for metric 2. Inspection of the raw data revealed an interesting trait. Implementations with small numbers of `Tags` and `Data` values tended to generate more states than transitions. This was not the case for larger combinations of `Tags` and `Data` values, where the gap between these two metrics closed rapidly; the number of transitions then overtook the number of states. Also of interest is the narrow range of the number of sub-processes in the implementation (metric 8), indicating that as the number of states increases, the architecture of the implementation remains fairly static; this was found to be a feature of Systems one and two, in complete contrast to Systems three and four (Chapter 7). Scatter plot analysis in later sections of this chapter illustrates this feature of Systems one and two in greater detail.

The minimum, maximum, median, mean and standard deviation CPU timings (in minutes) for the set of refinement checks are provided in Table 6.2. CPU timings are given since they give an indication of the computational effort required by the FDR model-checker and the program code used to generate the metrics.

The histogram for these timings is shown in Figure 6.2; it shows the limit of the machine's

	min.	max.	median	mean	std. dev.
System one	0.12	62.29	1.37	6.54	12.69

Table 6.2: Summary CPU statistics for System one

virtual memory to cope with the problem to be around sixty-two minutes (this was the refinement check for the five `Tags`, three `Data` values configuration).

Each refinement check (and hence each bar on the histogram) represents a `Tag` value and the associated set of `Data` values for that particular `Tag` (we will henceforward call this a *family*). Specifically, the first family comprises one `Tag` and one `Data` value, one `Tag` and two `Data` values, one `Tag` and three `Data` values, etc. until the virtual memory of the machine is exhausted. The second family comprises two `Tags` and one `Data` value, two `Tags` and two `Data` values, etc. until again the virtual memory of the machine is exhausted; this pattern is repeated up to six `Tags` and one `Data` value. (The above mode of computation of the refinement numbers obtains throughout the thesis). The histogram thus reflects six different families (one per `Tag`). The limit of each family for each set of refinement checks is given in Table 6.3.

Refinement number	Tags	Data values
1 - 10	1	10
11 - 19	2	9
20 - 29	3	10
30 - 33	4	4
34 - 36	5	3
37	6	1

Table 6.3: Limits of refinement families

We note that for a low number of `Tags`, i.e., one, two and three, an artificial limit was placed on the number of refinement checks undertaken. It was feasible to extend the one `Tag` relationship to a very large number of `Data` values for example, but this would add very little to the overall analysis. Most noticeable from Figure 6.2 is the nearest timing to the

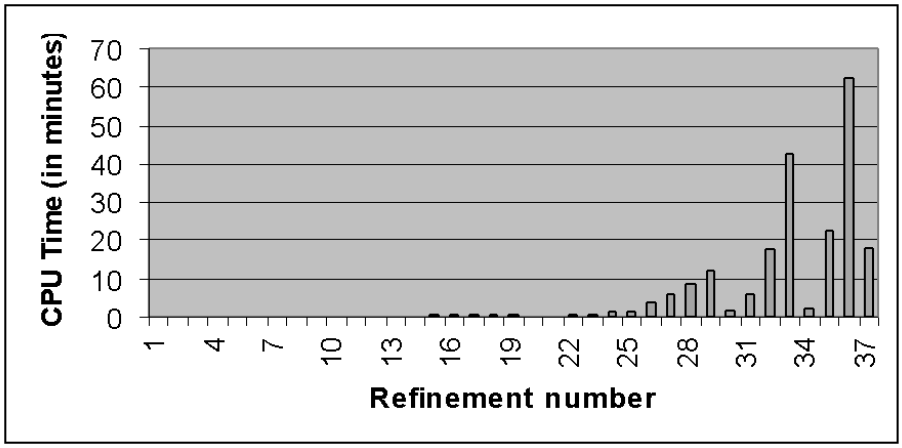


Figure 6.2: Histogram of CPU timings

limit of sixty-two minutes (i.e., forty-two minutes) applicable to the four `Tags` and four `Data` values configuration. The next highest timing applied to the configuration for five `Tags` and two `Data` values taking around twenty-two minutes to complete. This illustrates the nature of System one, in terms of the time it took to generate the twelve metrics as the state space was increased. The underlying structures (e.g., digraphs) representing the processes grew large very quickly.

In the sequel we will analyse experiments which were carried out for a *varying* number of refinement checks; this was dictated by various factors such as the virtual memory of the machine, the size of the stack, the application domain and whether we investigate refinement or non-refinement. On occasion, where it is manifestly obvious, tables like Table 6.3 will be omitted. (Note that each refinement number induces a refinement check.)

6.3.4 Refinement scatter plots

Choice of scatter plots

Since there are twelve metrics in the proposed set, a choice had to be made as to which scatter plots to produce for analysis from the one hundred and thirty-two possible combinations

of pairs of metrics. Since we are investigating the fidelity of an implementation with its specification, the choice of scatter plots was influenced strongly by those pairs of metrics which would shed some light on the relationship between specification and its implementation. The scatter plots chosen represent pairs of metrics which it was hoped would reveal most about the refinement process and provide an insight into the fidelity of an implementation with its specification; as such, there is strong emphasis placed on states, actions, transitions, non-divergent states and hidden behaviour.

We note that a robust correlation coefficient such as Kendall's τ could have been computed for each pair of metrics to identify relationships worth further exploration. Exploratory analysis for our selection of pairs of metrics did reveal mostly significant correlations. However, the nature of the set of metrics collected, for example, number of states and transitions in the implementation (between which there is an obvious intuitive relationship) meant that τ coefficients only confirmed what was evident from the scatter plots. In other cases, a correlation coefficient could not be computed; for example, where the values of one metric (for all refinement checks) were identical and the values of the other metric (for the same refinement checks) were wildly fluctuating. While valuable in certain cases, we feel that our approach for investigating relationships is just as valid as an approach based on τ correlation coefficients.

We also note that there was an element of *prospecting* in the choice of scatter plots in the hope that this may have revealed interesting, yet unexpected relationships[1]; as it turned out, a number of unexpected relationships did, in fact, arise from the analysis. All pairs of metrics and hence scatter plots were considered, but only those of interest/relevance to

[1]The same can be said of the metrics themselves. We do not claim that they are the definitive set for characterising fidelity. In the worst case, all that might be learnt is that the set of metrics is inappropriate for analysing fidelity. However, the hope is that they will illuminate some aspects of the refinement process and hence of fidelity.

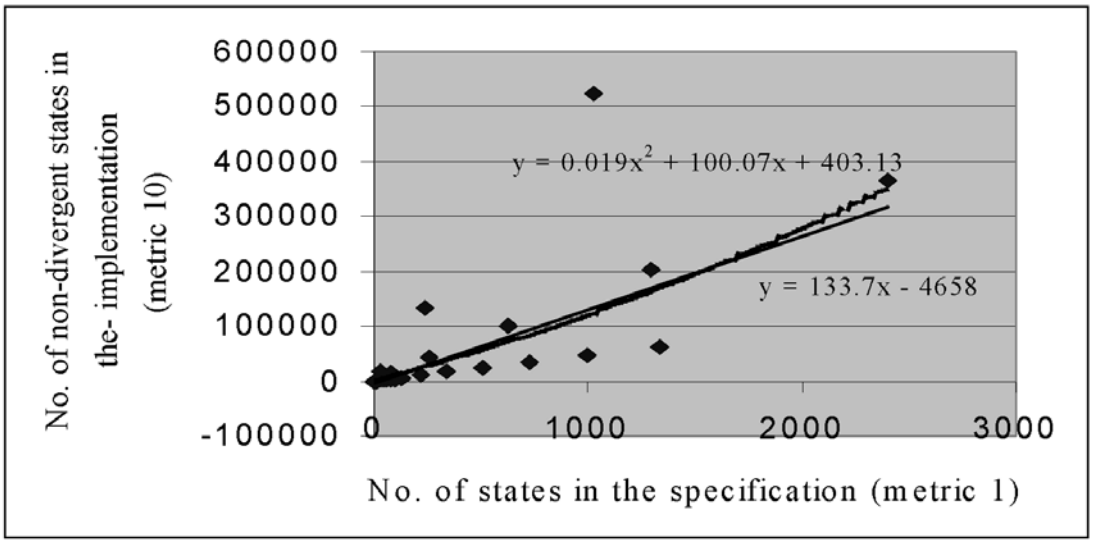


Figure 6.3: Metric 10 versus metric 1

the analysis were used in this thesis. As well as the equations of the lines and curves fitted around the scatter plots, we also present, where appropriate, the relevant R-squared values (*Rsv*) (see Appendix F) indicating the *goodness-of-fit* of that line or curve.

Finally, we note that, in order to avoid confusion for the reader, for System one the specification is called `Spec` and for System two the specification is called `SPEC`. In the sequel, each of the scatter plots is titled with the two metrics it represents on the appropriate axes.

Figure 6.3 shows the relationship between the number of non-divergent states in the implementation (metric 10) and the number of states in the specification (metric 1). A *best fit* straight line and quadratic curve are given; for both $Rsv = 0.47$.

Interestingly, the analysis appeared to reveal an anomaly in the data when plotting metric 10 versus metric 1 for each family. One family showed a sudden increase beyond a certain point implying that the number of non-divergent states was not constant. Inspection of the raw data revealed that this phenomenon in fact represented two families; namely, the 1 `Tag` n `Data` value configuration, where states in the specification merely increase by one on each new refinement check and then continuation of this line for the 3 `Tag` n `Data` value

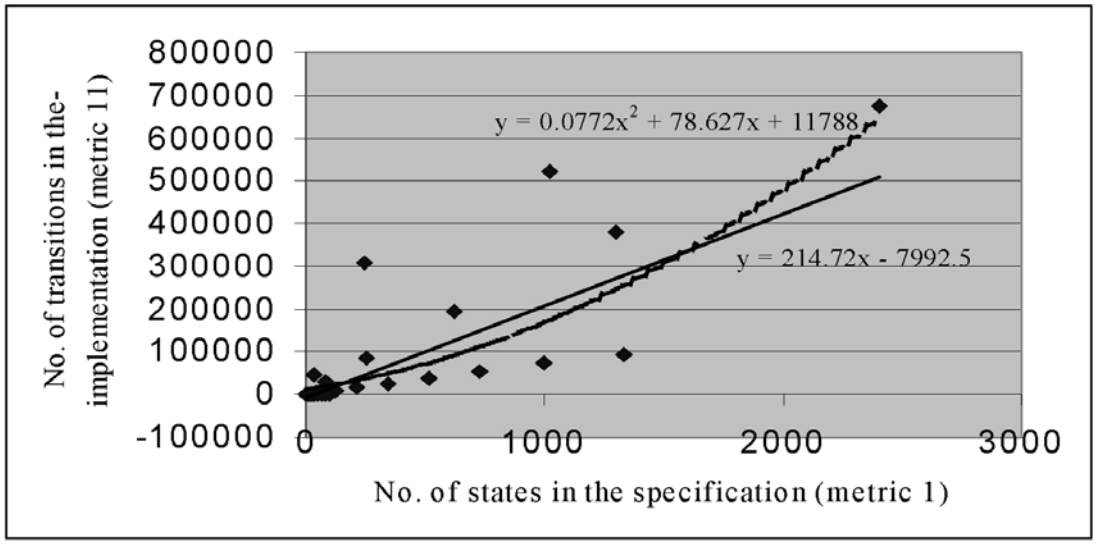


Figure 6.4: Metric 11 versus metric 1

configuration (hence the sudden rise). The overall trend in the data, however, is given by the monotonically increasing *best fit* quadratic curve, indicating an increasing rate in the growth of non-divergent implementation states as the number of specification states increases. On the empirical evidence obtainable, the gradient of this curve is gradually increasing.

A similar trend is exhibited in Figure 6.4, plotting metric 11 versus metric 1. The Rsv for the quadratic is 0.63 (compared with 0.58 for the straight line). This relationship is not entirely unexpected, since the increase in the number of transitions in the implementation should rise at an increasing rate with increases in the number of states in the specification. It is worthwhile noting at this point that the number of 'transitions' metric includes all handshake (or tau) transitions and hence has a higher upper limit than the corresponding number of non-divergent states in the implementation. Mathematical analysis supported the observation that the quadratic curve in Figure 6.4 does indeed have a steeper gradient than that of Figure 6.3; transitions are therefore generated more quickly than non-divergent states for the same number of specification states in System one.

The relationship between metric 10 and metric 2 is shown in Figure 6.5(a). This is a

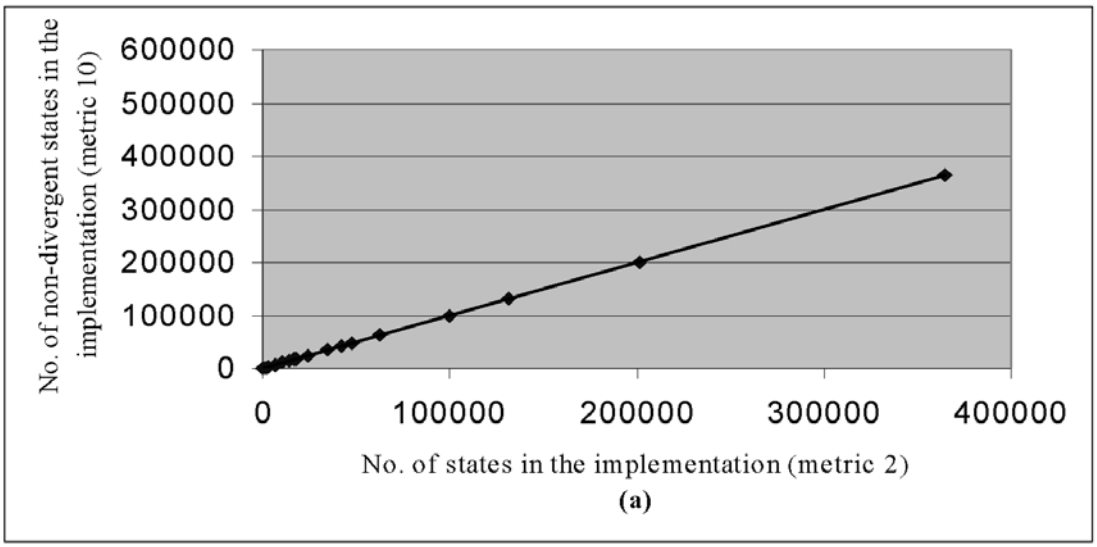
45-degree straight line, i.e., the number of non-divergent states is equal to the number of states in the implementation, by definition of refinement as was observed after Table 6.1.

Inspection of the raw data revealed that for Figure 6.5(b) there is a tendency for the number of transitions in the implementation to increase at a slightly higher rate than the number of states in the implementation (as the size of the datatypes, i.e., `Tag` and `Data` values is increased). Evidence of this is given by the monotonically increasing *best fit* quadratic curve (dashed) which meets the straight line at the point where $x = 364440$ implementation states; the relationship between these two metrics, i.e., 11 and 2 is clear: implementation transitions increase at a faster rate than implementation states. The straight line in this case is a better fit (with $Rsv = 0.99$) compared to the quadratic (with $Rsv = 0.68$).

This trend is more evident when looking at the corresponding histograms of Figure 6.6. These show that, moving from a configuration of five `Tags`, one `Data` value (refinement number 34) to a configuration of five `Tags`, two `Data` values (refinement number 35) sees the number of non-divergent states rise from 201048 to 364640. For the same change of configuration the number of transitions rises from 380832 to 676128.

The relationship between transitions and states (Figures 6.4 and 6.5(b)) for System one would seem to reflect the nature of the application domain and its dependence on communication between CSP processes. In other words, protocol type applications tend to contain a high proportion of handshake operations, a feature which was found to a lesser extent in System two, and absent in Systems three and four.

Figure 6.7(a) shows the number of harmless partially divergent states in the implementation against the number of states in the specification (metric 12 against metric 1). Again, the trend is for the number of harmless partially divergent states to rise at an increasing rate as `Tag` and `Data` values are combined. This relationship (see dashed *best fit* quadratic curve



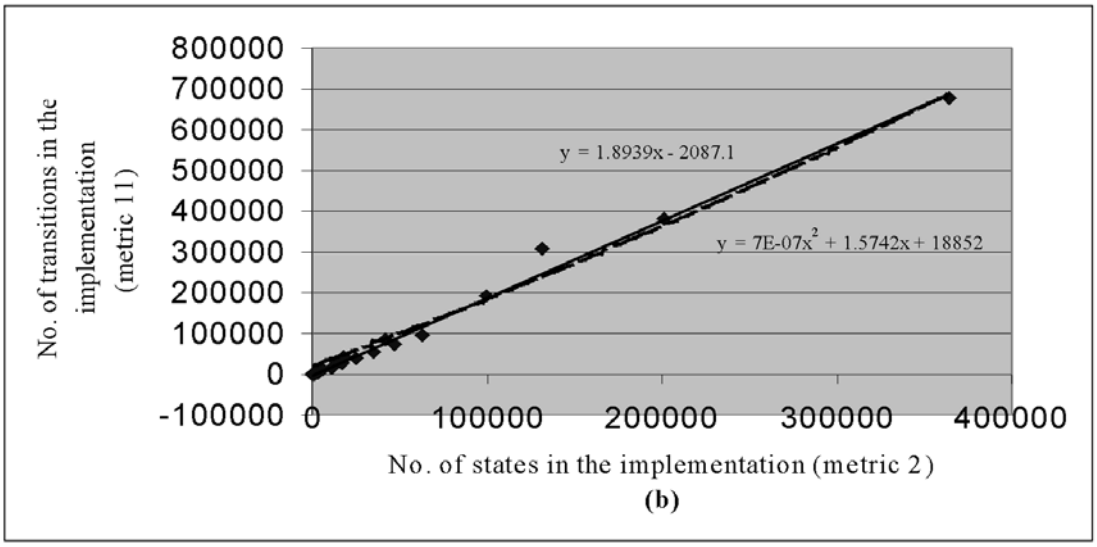


Figure 6.5: Metric 10 versus metric 2 and metric 11 versus metric 2

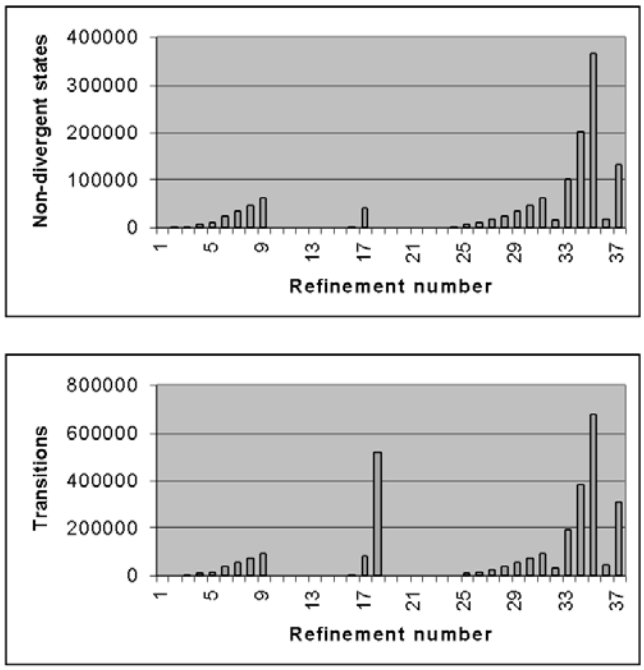
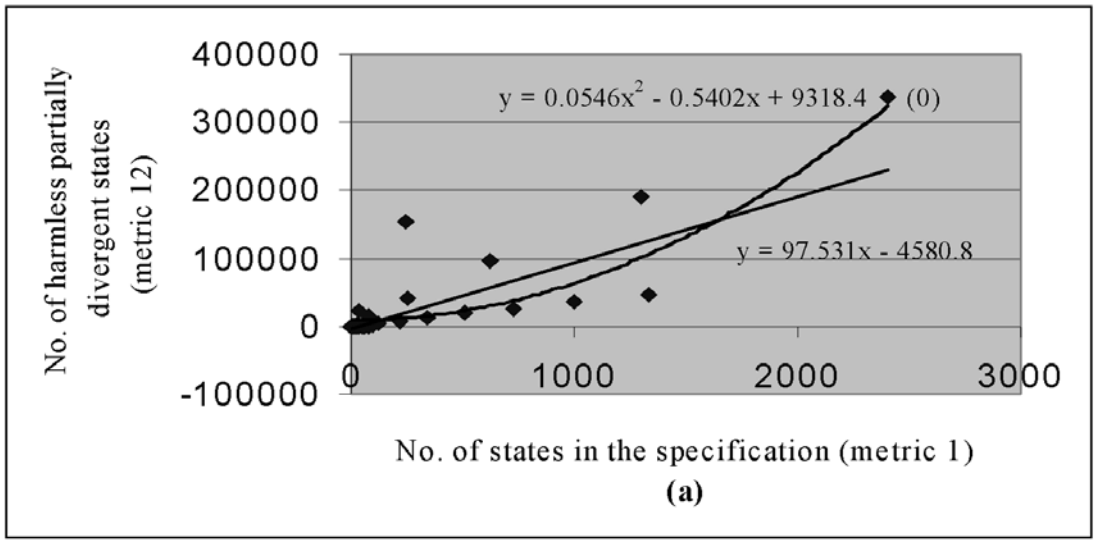


Figure 6.6: Histograms showing increase in non-divergent states and transitions in the implementation across refinement checks

in Figure 6.7(a)) resembles that in Figure 6.4, suggesting a strong relationship between the number of transitions in the implementation (metric 11) and harmless partially divergent states (metric 12). The outermost value (0) on this plot is for the configuration of five `Tags`, three `Data` values. Again, the inability to go beyond this point was due to the limitation imposed by the virtual memory of the machine on which the experiments were undertaken. The *Rsv* value for this curve is 0.72; for the straight line, the *Rsv* is 0.61.

From the summary metrics in Table 6.1, it can be deduced that the number of harmless partially divergent states (metric 12) is consistently less than the number of states in the implementation (metric 2). The exact relationship is given in Figure 6.7(b). This figure shows the quadratic relationship (dashed line) and the linear relationship to virtually overlap. The *Rsv* value for both is 0.99.

The scatter plot in Figure 6.8, representing sub-processes in the implementation (metric 8) versus actions in the implementation (metric 4), shows no discernible pattern. The *Rsv*



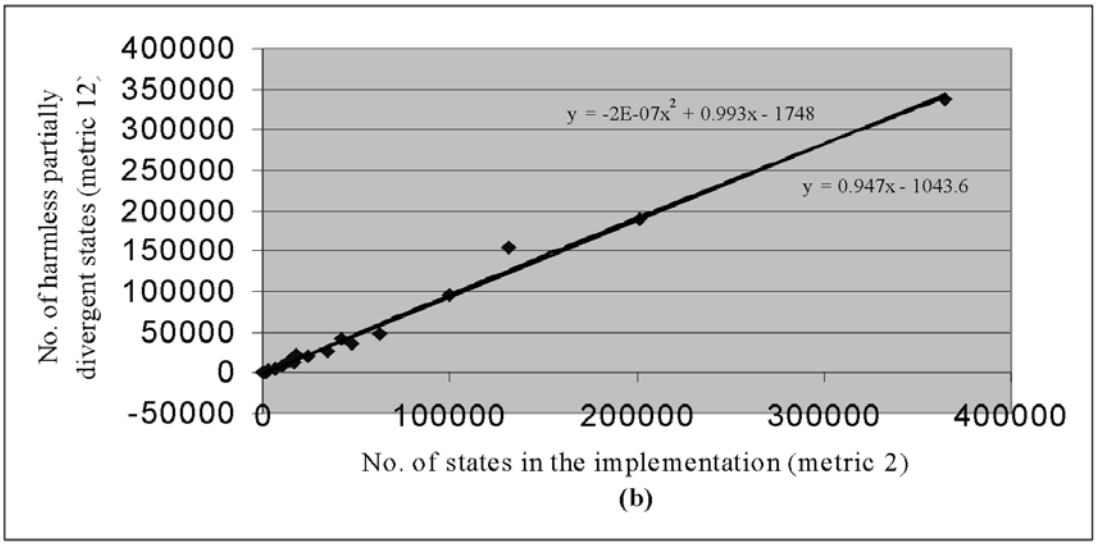


Figure 6.7: Metric 12 versus metric 1 and metric 12 versus metric 2

for *best fit* line is 0.13.

However, inspection of the original data revealed that the number of sub-processes in some cases remained static when the number of actions in the implementation was increased, and in other cases increased. This is better illustrated with a bar chart; in Figure 6.9, the lighter bars in the histogram represent actions and the darker bars represent sub-processes.

Both the scatter plot and the histogram seem to suggest that within development of System one, there are some actions that do not affect the number of sub-processes, whilst

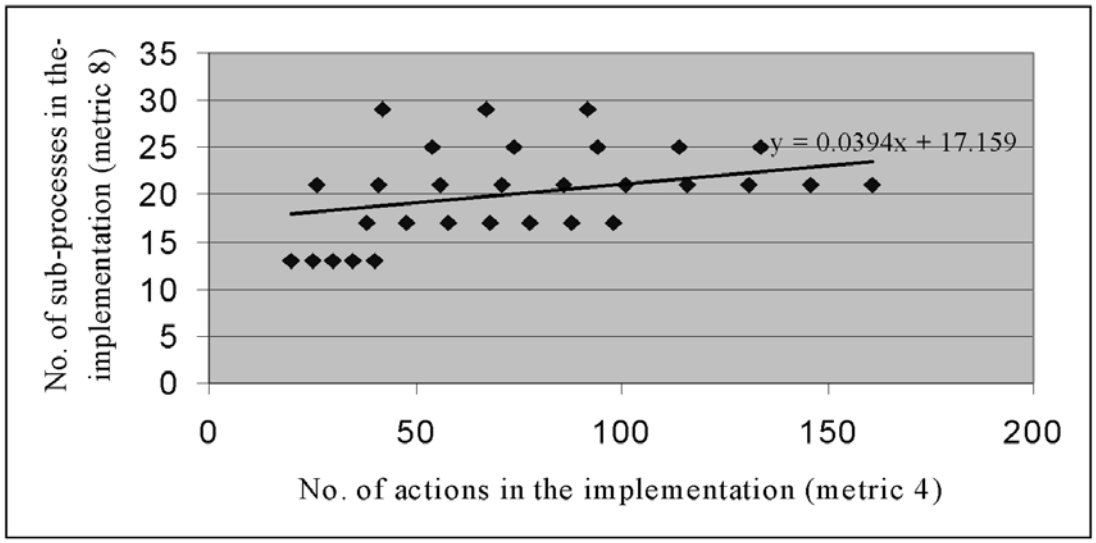


Figure 6.8: Metric 8 versus metric 4

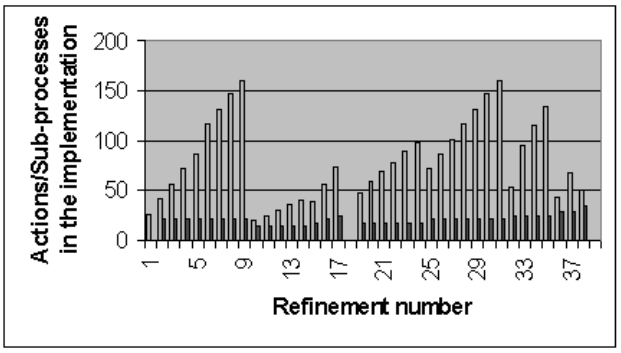


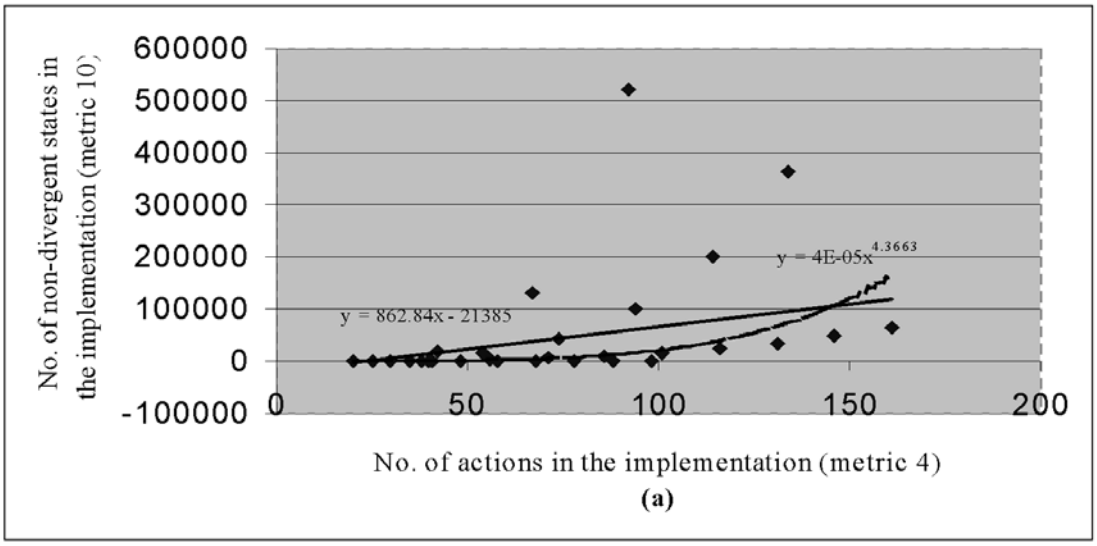
Figure 6.9: Histogram of actions and sub-processes in the implementation across refinement checks

the addition of other actions causes the number of sub-processes generated to increase. Interestingly, this trend was also noticed for System two. (We note that generation of sub-processes through recursive invocation was a major inhibiting factor in our analysis of Systems three and four.) In the case of System one, the reason behind this feature was due to the type of action being added. In other words, it is due to the difference between adding `ack` actions, which contained no variables and hence added little behaviour to the overall process, and adding actions of the form `left.x.y` which generated a variety of sub-processes

depending on the size of `x` and `y` datatype values. The `ack` actions represent the interface between the sub-processes rather than intrinsic features of the sub-processes themselves and hence cause fewer sub-processes as part of the hidden behaviour of a process.

Scatter plots in Figure 6.10 reinforce the relationship between non-divergent states and transitions in the implementation with reference to metric 4. Key to the increase in the number of non-divergent states and transitions is the increase in the number of actions in the implementation. We observed the same phenomenon with reference to metric 1 (Figures 6.3 and 6.4) and metric 2 (Figures 6.5(a) and 6.5(b)). (The *Rsv*'s show the quadratic curves to be the *best fit* in each case (0.67 and 0.63 for (a) and (b), respectively)).

Figure 6.11(a) shows an interesting feature, and reiterates a point made earlier, but then it was made in terms of the number of actions in the implementation (Figure 6.8). As the number of states in the implementation increases, the number of sub-processes in certain cases remains static, and in other cases increases. The general trend, however, is for the number of sub-processes to increase initially and then flatten out beyond a certain point which would indicate the point at which adding behaviour (in terms of extra states) may not necessarily cause more sub-processes to be generated. Again, the key to this question (as was only later discovered when looking at System two) was in the different types of action embedded within a process, since different actions were capable of generating different numbers of sub-processes. A CSP process predominantly containing `ack` commands seems to generate fewer sub-processes than one using and manipulating data values (such as `left.Tag.Data`); the role of `ack` commands is to synchronise communication between sub-processes. They do not feature in the more involved workings of those sub-processes. Inspection of the raw data revealed the number of sub-processes to stabilise at around 29 (Figure 6.11(a)). Beyond this point, due to the limitations imposed by the virtual memory



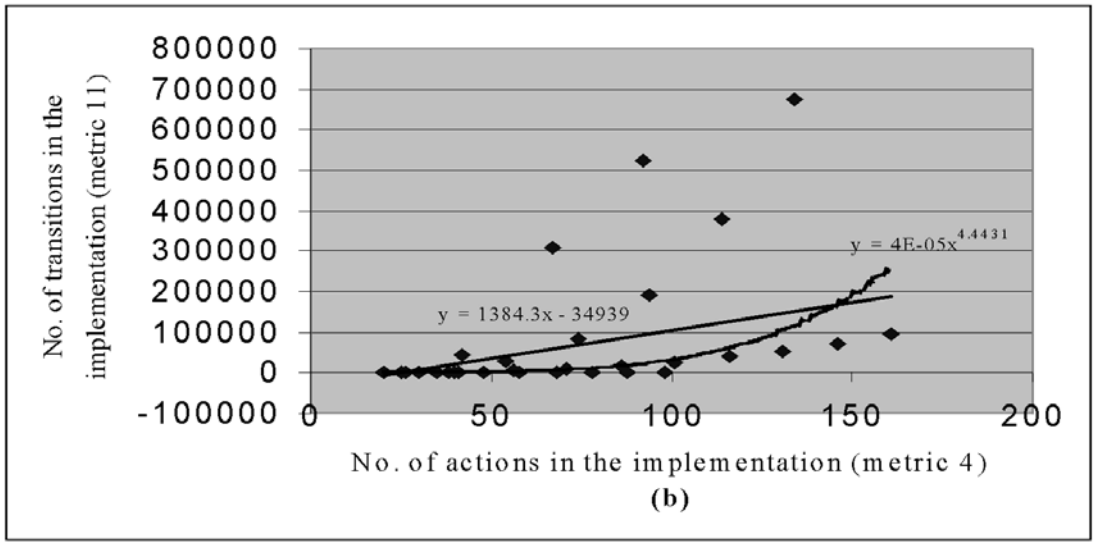
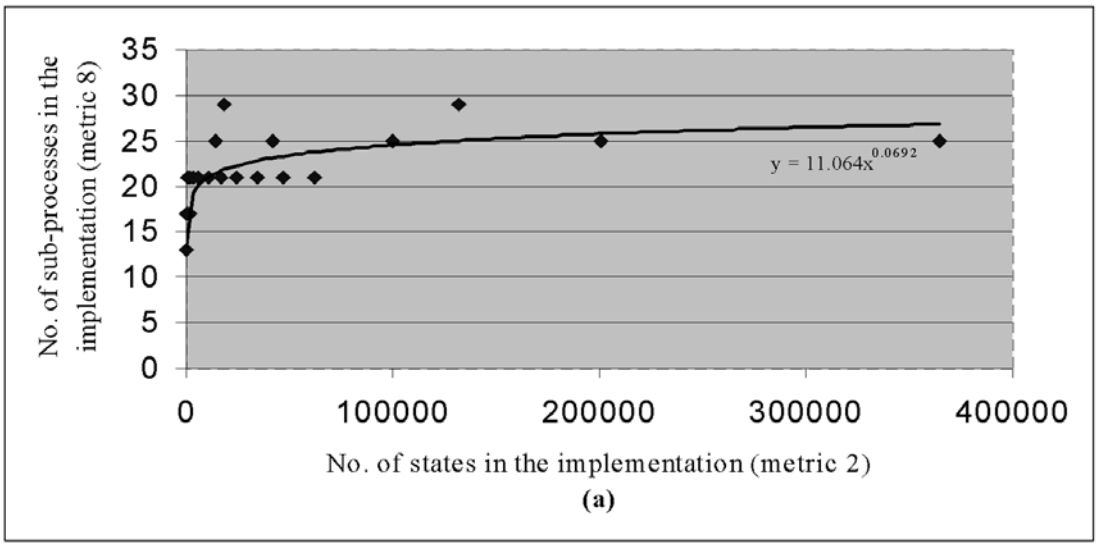


Figure 6.10: Metric 10 versus metric 4 and metric 11 versus metric 4

of the machine, it was impossible to increase this number. (We note that, as a result of the recursive nature of Systems three and four, a large number of sub-processes were generated in a similar scenario. However, as will be explained in the following chapter, this caused its own problems.)

Figure 6.11(b) shows the scatter plot of metric 8 against metric 5. No immediate relationship is clear which seemed unusual at first, since sub-processes should encapsulate hidden behaviour. After careful thought, it became evident that the nature of this relationship is



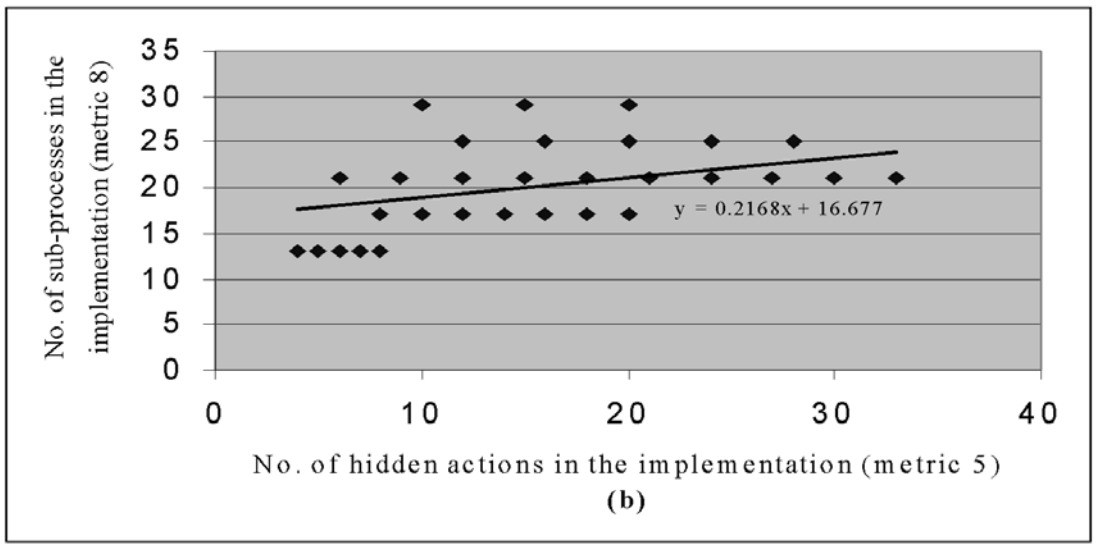
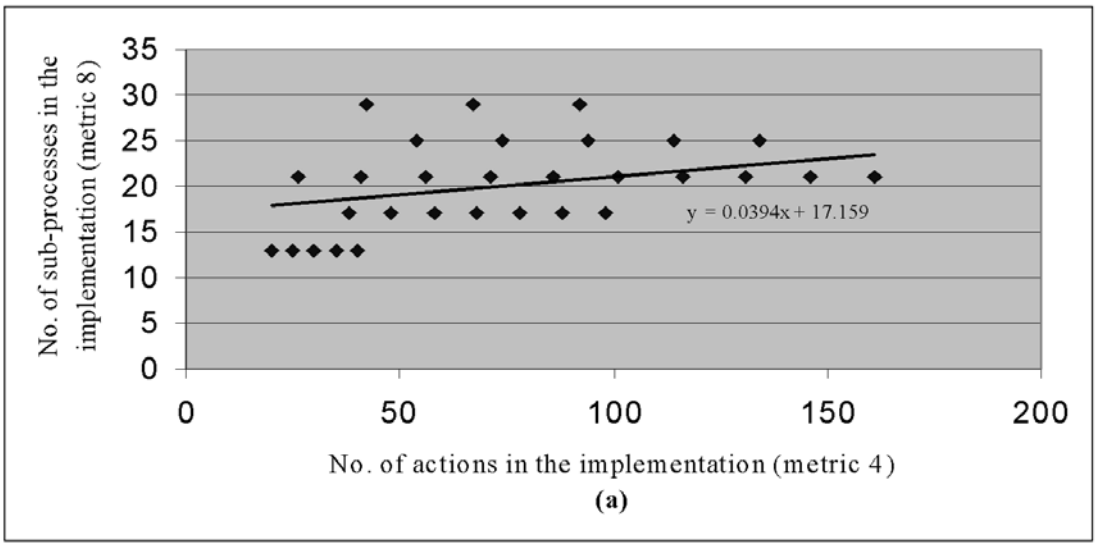


Figure 6.11: Metric 8 versus metric 2 and metric 8 versus metric 5

developer-oriented, i.e., the extent of hiding in a CSP system is down to the developer and will hence vary. A similar phenomenon is shown in Figure 6.12(a), but more actions are evident on the x-axis for each sub-process when compared to Figure 6.11(b).

Finally, Figure 6.12(b) shows the relationship between the number of states in the implementation and the number of states in the specification. The number of states in the implementation increases at a higher rate than the number of states in the specification; the Rsv for the quadratic curve is 0.80, and for the straight line is 0.69.



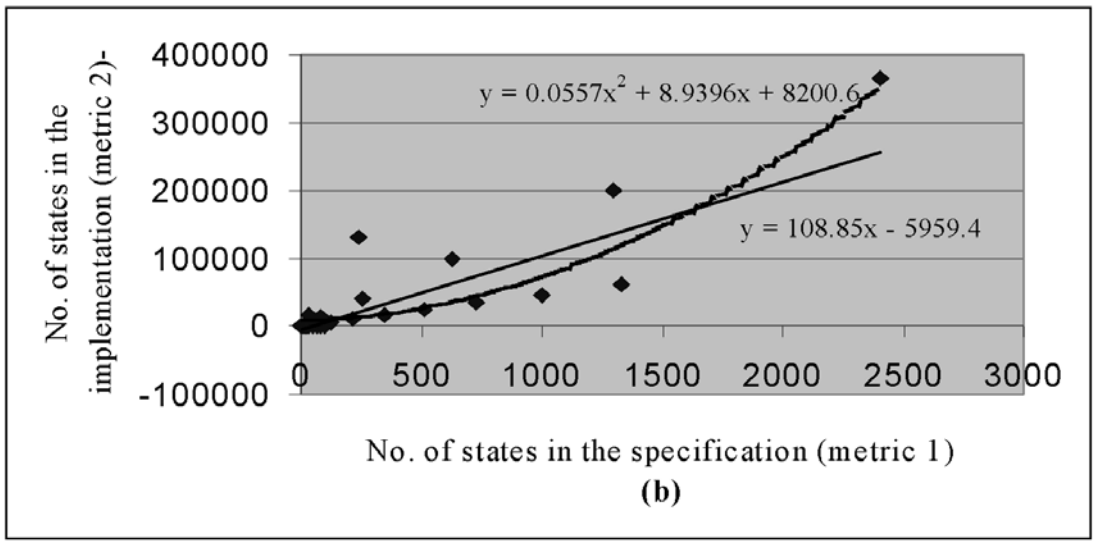


Figure 6.12: Metric 8 versus metric 4 and metric 2 versus metric 1

6.3.5 Refinement of System one: discussion

A number of conclusions can be drawn from refinement within the context of the multiplexed buffer application. Perhaps not unexpectedly, the number of states and transitions increase at an accelerating rate as the number of `Tags` and `Data` values is increased. This can be seen clearly from the scatter plots (Figures 6.4 and 6.5(b)). However, because of the dependence on handshake operations within bit-oriented applications, the number of transitions (metric 11) rises at a faster rate than that of states. This is interesting for several reasons:

Firstly, when we come to look at injecting faults into the multiplexed buffer system, those injected at the interface level (at the point where a sub-process communicates with another) may propagate further and have different features to those faults injected at other parts of the process.

Secondly, from a modularity perspective, an optimum level of coupling may exist between processes, which minimises the propagation of faults but maximises comprehensibility arising from proper use of coupling. Although beyond the scope of this thesis, a useful exercise might be to look at fault propagation, with varying numbers of sub-processes, investigated from an empirical viewpoint.

From the analysis of the multiplexed buffer application it would also appear that, in certain circumstances, the number of sub-processes remains static when the behaviour of the application (in terms of actions) is increased. This may reflect the robust nature of the architecture of a multiplexed buffer system; in other words the addition of any extra behaviour added through actions (e.g., `left`, `right` and `ack`) does not necessarily change the structure of the application, and when it does, it does so only very marginally. However, it must be noted that some types of added behaviour did generate more states (and potentially more sub-processes) than others. A good example is the difference in the potential for added behaviour between actions such as `left.x.y` and a simple `ack` action.

Finally, no relationship was found between sub-processes and hidden actions, which at first seemed counter-intuitive, but after much thought was a reasonable conclusion to draw based on knowledge of the other three systems. The extent of hiding in the implementation is the responsibility of the developer and no immediate relationship between sub-processes and hiding should be expected.

6.3.6 System two: the alternating bit protocol

The full code for the alternating bit protocol system is given in Appendix B. The following description of the main parts of the CSP code is taken on the whole from [For97].

> This is the initial example of a set which makes use of a pair of media which are permitted to lose data, and provided no infinite sequence is lost will work independently of how lossy the channels are. They work by transmitting messages one way and acknowledgments the other. The alternating bit protocol provides the most standard of all protocol examples.

In the sequel, only the processes relevant to the refinement checks undertaken are listed. These are `SPEC`, `PUT`, `GET`, `SEND` and `RECV`, followed by the refinement checks carried out.

```
Channels and data types
left and right are the external input and output.
a and b carry a tag and a data value.
c and d carry an acknowledgment tag.
(In this protocol tags are bits.)

            a  PUT  b
left        /    \    right
------> SEND       RECV ------>
            \       /
            d  GET  c

DATA = {2,3}

channel left,right : DATA
channel a, b : Bool.DATA
channel c, d : Bool
{-
  The overall specification we want to meet is that of a buffer.
-}
SPEC = let
{-
  The most nondeterministic (left-to-right) buffer with size bounded
  by N is given by BUFF(<>, N), where
-}
BUFF(s, N) =
```

```
  if null(s) then
    left?x -> BUFF(<x>, N)
  else
    right!head(s) -> BUFF(tail(s), N)
    []
     #s < N & (STOP |~| left?x -> BUFF(s^<x>, N))
{-
  For our purposes we will set N = 3 since this example does not introduce
  more buffering than this.
-}
within BUFF(<>, 3)

lossy_buffer(in, out, bound) =
  let
    B(0) = in?x -> out!x -> B(bound-1)
    B(n) = in?x -> (B(n-1) |~| out!x -> B(bound-1))
  within B(bound-1)

PUT = lossy_buffer(a, b, 3)
GET = lossy_buffer(c, d, 3)
{-
  The implementation of the protocol consists of a sender process and
  receiver process, linked by PUT and GET.
-}
SEND =
  let
    Null = 99 -- any value not in DATA
    S(v,bit) =
      (if v == Null then left?x -> S(x, not bit) else a!bit!v -> S(v, bit))
      []
      d?ack -> S(if ack==bit then Null else v, bit)
  within S(Null, true)

RECV =
  let
    R(bit) =
      b?tag?data -> (if tag==bit then right!data -> R(not bit) else R(bit))
      []
      c!not bit -> R(bit)
    {-
      The first message to be output has tag false, and there is no pending
      message.
    -}
  within R(false)

make_system(receiver) =
  make_full_system(SEND, PUT|||GET, receiver)

make_full_system(sender, wiring, receiver) =
  sender[|{|a,d|}|](wiring[|{|b,c|}|]receiver)\{|a,b,c,d|}
```

```
DIVSYSTEM = make_system(RECV)

NDC(M) =
  let
    C(n) =
      if n==0 then
        c?_ -> C(n+1)
      else if n==M then
        b?_ -> C(n-1)
      else
        c?_ -> C(n+1) |~| b?_ -> C(n-1)
         within C(M/2)

       RCVL = modify_receiver(LIMIT(3))
       RCVN = modify_receiver(NDC(4))

       -- and the checks of the respective systems against SPEC

       assert SPEC [FD= make_system(RCVL)
       assert SPEC [FD= make_system(RCVN)
```

Twenty refinement checks were completed for the alternating bit protocol. We note that, again, because of the limitations imposed by the machine on the amount of available virtual memory, it was impossible to run the same number of refinement checks for Sytem two as were carried out for System one. Nine of the twenty refinement checks were as a result of increasing the value of the parameter of the process `NDC` at selected intervals up to the point at which the virtual memory of the machine had been exhausted (this value was realised when the parameter to `NDC` was set at `M` $= 200$). The remaining eleven refinement checks were produced as a result of increasing the parameter of the process `LIMIT`, set at the value three in the previous code, again, up to a limit imposed by the virtual memory of the machine (in this case, value 600).

Table 6.4 shows summary metrics for the twenty refinement checks. Compared with the same summary metrics for System one, a number of notable differences arise. To start with, the number of actions in the implementation (metric 4) remains static; so too does the number of hidden actions in the implementation (metric 5) and the number of sub-processes

in the implementation (metric 8). This is despite the fact that the size of the state space was being increased with successive changes to the parameters of `LIMIT` and of `NDC`. The upper limit on the number of states in the implementation is comparable to that for System one. However, the number of transitions for System two far exceeds that of System one. The reason for this is simple. The handshake (or tau) action which contributes to the number of transitions is greater in System two than in System one.

Inspection of the code for System two revealed that the logic in the alternating bit protocol was far more involved than in the multiplexed buffer system in terms of error checking and case testing. This accounts for the increasing number of states in the implementation (metric 2). Other features such as the complexity of the code (see Appendix B) also contributed to the trend for a larger number of transitions in this system vis-*à*-vis System one.

Metric	min.	max.	median	mean	std. dev.
1 Number of states in the specification	27	27	27	27	0
2 Number of states in the implementation	3327	452112	28599	98274	138743.60
3 Number of distinct actions in the specification	6	6	6	6	0
4 Number of distinct actions in the implementation	18	18	18	18	0
5 Number of hidden actions in the implementation	12	12	12	12	0
6 Number of visible actions in the implementation	6	6	6	6	0
7 Number of sub-processes in the specification	0	0	0	0	0
8 Number of sub-processes in the implementation	9	9	9	9	0
9 Number of additional actions in the implementation	12	12	12	12	0
10 Number of non-divergent states in the implementation	3327	452112	28599	98274	138743.60
11 Number of transitions in the implementation	8441	1317380	77689	273007	387065.10
12 Number of harmless partially divergent states	4220	658690	38844	136504	193532.60

Table 6.4: Summary metrics for System two

The CPU timings for System two are contained in Table 6.5. Of interest is the maximum CPU timing (at just over sixty-four minutes), generating 452112 states in the implementation. For System one, 364440 states were generated in sixty-two minutes, suggesting that System two is capable of generating states at a faster rate than System one. One explanation for this could be that, unlike System one, the number of sub-processes in the implementation

for System two remains static across refinement checks (a striking feature of the application). It thus becomes much easier to generate extra states due to this stability. Thus one noticeable difference between System one and System two is in the architectures of each system and their ability to generate states.

	min.	max.	median	mean	std. dev.
System two	0.18	64.06	1.26	8.91	17.13

Table 6.5: Summary CPU statistics for System two

The histogram for the timings is shown in Figure 6.13. A similar trend is apparent as in the multiplexed buffer timings. As noted above, the maximum CPU timing occurs at just over sixty-four minutes. The next highest timing stands at just over forty-two minutes, and the next highest at just over thirty minutes. The preceding timing to that was just over thirteen minutes. This sequence of timings seems to suggest that as the size of the state space is increased, the time taken to perform the refinement check is increasing, but at a decreasing rate[2].

This feature may again be due to the architecture of the system as it becomes easier to generate states around a stable architecture. More empirical studies would need to be undertaken, however, before this phenomenon could be fully explained.

6.3.7 Scatter plots

Choice of scatter plots

The choice of scatter plots is intended (as for System one) to highlight the important features of processes for the model of fidelity we introduced in Chapters 3 and 4. The choice of scatter

[2]Later in this chapter, an interesting difference between the implementation and specification state digraphs for the two systems is identified.

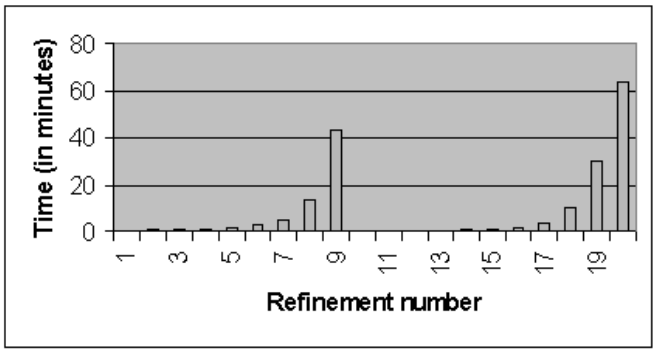
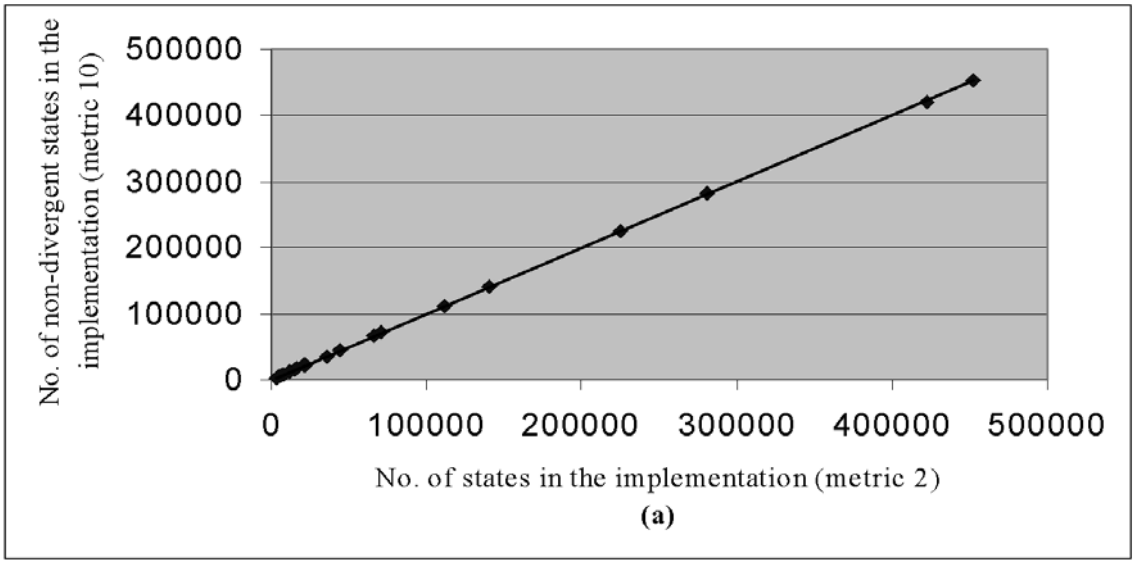


Figure 6.13: Histogram of CPU timings

plots was also influenced by the need to compare features with those found in System one; as such, many of the scatter plots used for System one were also produced for System two.

Scatter plots in Figure 6.14 exhibit similar patterns to those for the multiplexed buffer (Figure 6.5(a),(b)). Since the implementation refines the specification, the number of states in the implementation is equal to the number of non-divergent states (Figure 6.14(a)). However, unlike System one, where the straight line and quadratic curve showed some deviation, the two in Figure 6.14(b) are completely overlapping. This implies that transitions are generated at a slower rate in System two (compared with System one), relative to the number of states in the implementation. The Rsv values for the straight line and quadratic curve were both 0.99, illustrating the closeness of fit.

Figure 6.15(a) reflects the static nature of the sub-processes for the implementation (metric 8). The false appearance is of a single plotted value on the scatter plot (there are actually twenty plotted values on the same spot corresponding to the twenty refinement checks). It is interesting to note that increasing the value of the parameter `M` of `NDC`, as shown in the fragment of code below, does not increase the number of actions in the implementation be they hidden or visible (yet the number of states in the implementation does rise). Further investigation and tracing of the CSP process revealed a dependence of the system on the use



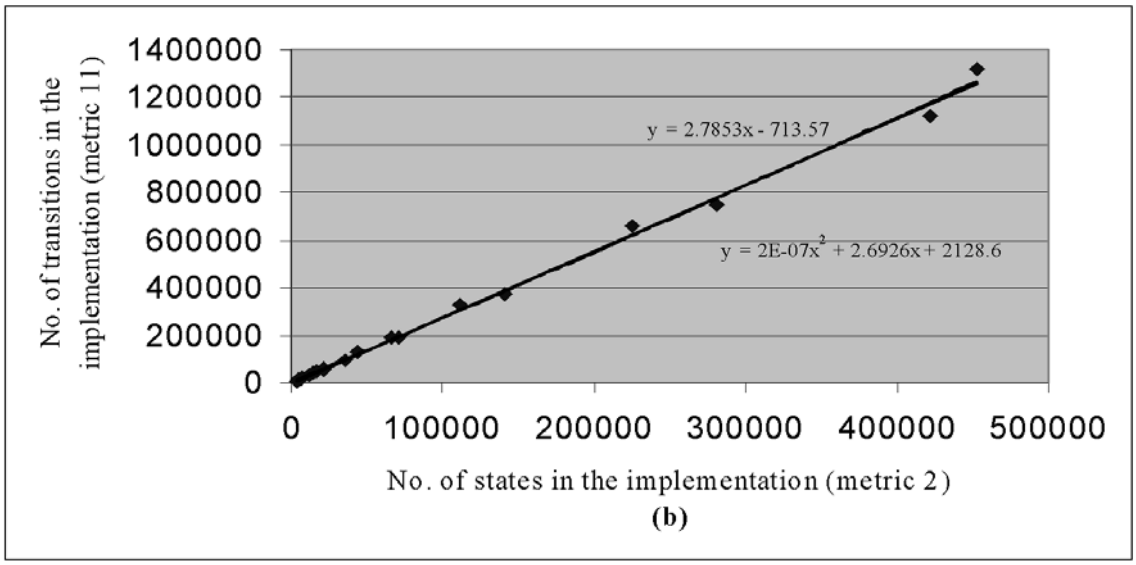


Figure 6.14: Metric 10 versus metric 2 and metric 11 versus metric 2

of parameter passing to build up the size of the state space; this is unlike System one, where communication of values was the key to its state space size and structure. This implies a different emphasis in the way the two systems were designed and coded and, more importantly, in the architecture of the two systems, as the empirical evidence seems to suggest. As an example of how highly parameterised System two is, consider the following CSP process (`NDC` fragment of System two) containing significant parameter passing:

```
NDC(M) =
  let
    C(n) =
      if n==0 then
        c?_ -> C(n+1)
      else if n==M then
        b?_ -> C(n-1)
      else
        c?_ -> C(n+1) |~| b?_ -> C(n-1)
  within C(M/2)
```

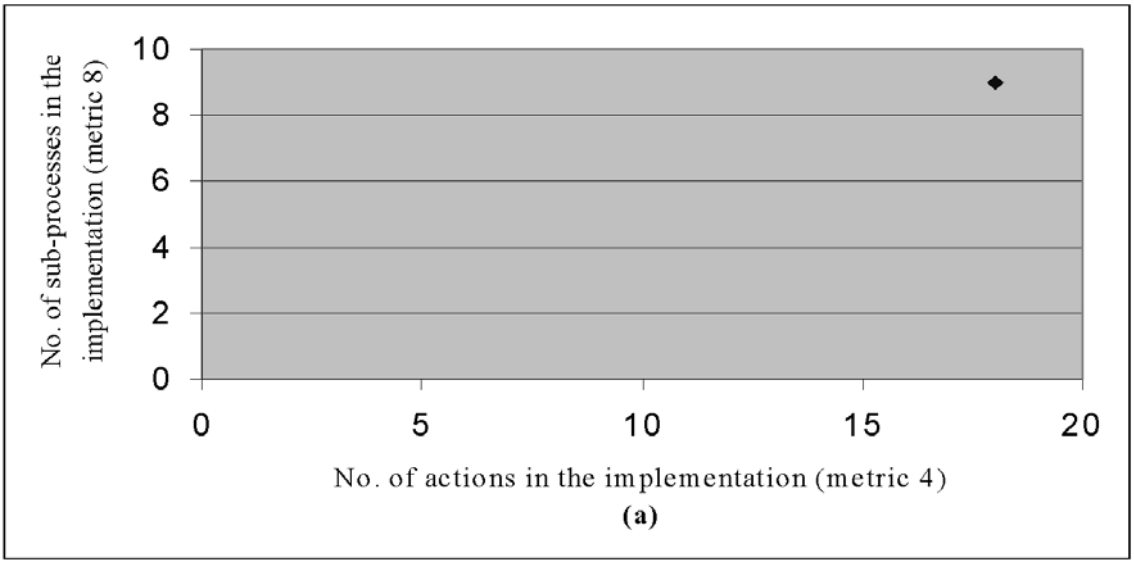
Such code is not found in System one. Figure 6.15(b) reinforces the static nature of the actions in the implementation.

Scatter plots for other combinations of metrics were also produced, but tended to reinforce the static nature of the number of states in the specification and the number of actions in the implementation; thus they have been omitted from this analysis.

6.3.8 Refinement of System two: discussion

A number of conclusions can be drawn from the empirical analysis of the alternating bit protocol.

Firstly, the specification was static in the sense that its number of states remained constant across refinement checks. This implies that the *gap* between the specification and implementation (in terms of the number of non-divergent states, transitions and other met-



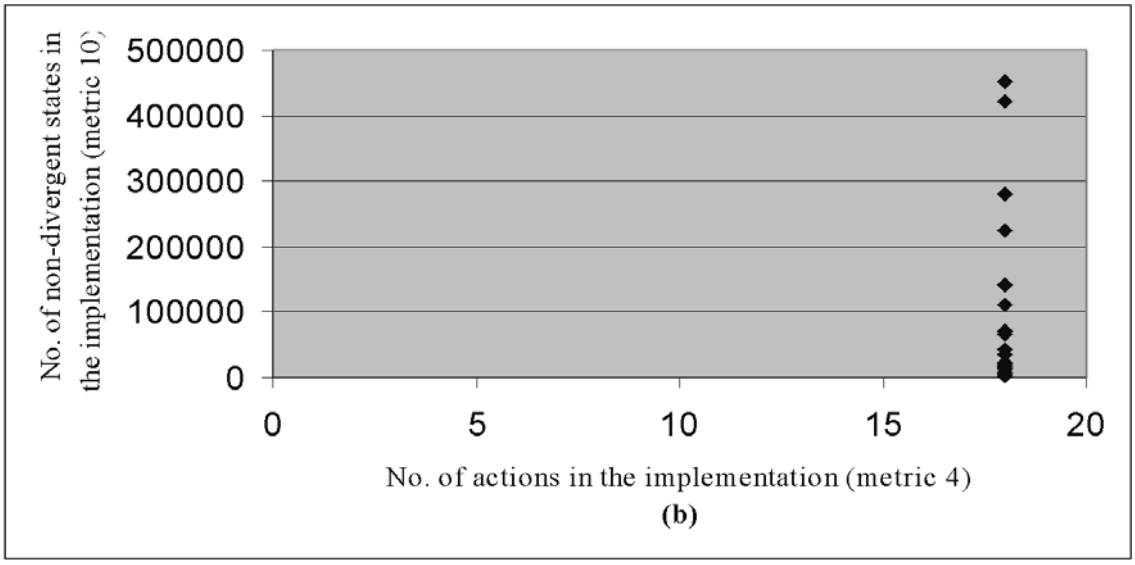


Figure 6.15: Metric 8 versus metric 4 and metric 10 versus metric 4

rics) widened as states were added to the implementation; refinement in this sense is different from that of System one. The extent of refinement given by our metrics seems to be dependent on how the specification is expressed and the ease with which the size of the state space of the specification can be increased. This could be viewed as a criticism of the metrics suggested. In defence of the metrics however, we feel that it is as worthwhile to analyse the size of this gap with a static number of specification states as with a changing number of specification states, since, at the very least, we obtain information on the behaviour of the implementation. We are also constrained by the nature of the application and the machine in terms of what refinement checks we can carry out; these in turn dictate the values of the metrics produced.

Secondly, the amount of parameter passing in a CSP system seems to have a large effect on the values of the metrics produced. System one was a standard CSP process which did no error-checking and contained very little coding logic; hence, channel-oriented value passing was used. On the other hand, System two employs error-checking and case-checking via parameter passing, manifesting itself in different state generation patterns.

We next re-visit Systems one and two in turn and look at non-refinement, i.e., when behaviour in the specification is not captured by the implementation. In this context, we take non-refinement to mean any situation where the FDR model-checker returns the result `xfalse` from the refinement check being undertaken; this indicates that the implementation does not refine its specification. Recall that (true) refinement is indicated by the FDR model-checker returning `true`. We pause briefly to consider the objective of looking at non-refinement.

Our objective, in considering non-refinement, is to see how the behaviour of the implementation with respect to the specification (reflected in terms of metrics values for each)

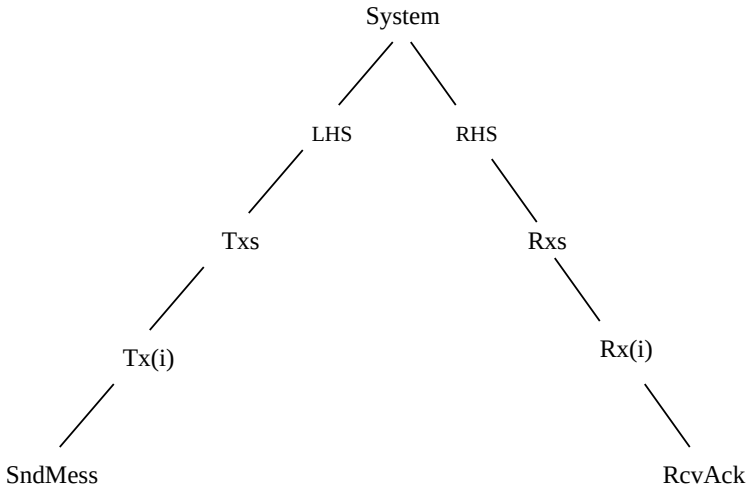


Figure 6.16: Tree decomposition of System one

changes as sub-processes are composed. In an environment where different sub-processes of the implementation may be allocated to different developers and then composed at a later date, it is useful for the developer to see which (if any) behaviour is common to specification and implementation at the early stages of development. To achieve this, we can choose any sub-process of the implementation and run a refinement check of that sub-process against the specification. In the context of this thesis, we can thus operate a middle-out policy towards our non-refinement (choosing sub-processes which already contain other sub-processes); a more realistic choice, however, would be a top-down policy, where the highest level (skeleton) sub-processes are checked first, then each of the sub-processes contained therein, until the lowest-level primitives are in place. Figure 6.19 shows the implementation of System one when decomposed into its constituent sub-processes in tree form. In the analysis of non-refinement which follows, we will look down one side of the tree (beginning with `LHS` then `Txs`, etc. and ending with `SndMess` before considering `RHS`).

6.4 Non-refinement of Systems one and two

For both the multiplexed buffer and the alternating bit protocol systems, a range of non-refinement checks were completed when the implementation did not capture its specification. In the following sections, we look at System one and develop our non-refinement analysis by looking at the largest sub-parts of this system, then the next largest sub-parts and so on, until the decomposition is complete, i.e., we arrive at CSP primitives. We therefore begin by looking at the features of `LHS`.

6.4.1 Multiplexed buffer: non-refinement

Process one: LHS

A set of eleven non-refinement checks were carried out on the process `LHS` when run against `Spec`. The limits for these refinement checks are shown in Table 6.6.

Refinement number	Tags	Data values
1 - 3	3	3
4 - 7	4	4
8 - 11	5	4

Table 6.6: Limits of refinement families

The CPU timings (in minutes and seconds) are given in Table 6.7.

	min.	max.	median	mean	std. dev.
System one	0.16	7.41	1.48	2.68	2.60

Table 6.7: Summary CPU statistics for System one non-refinement

Scatter plots were then produced to evaluate `LHS` against `Spec` as the specification. The scatter plots in Figure 6.17 show that `LHS` has one non-divergent state before a break down in the expected behaviour of the implementation is found. That is, in our case, the `LHS` CSP

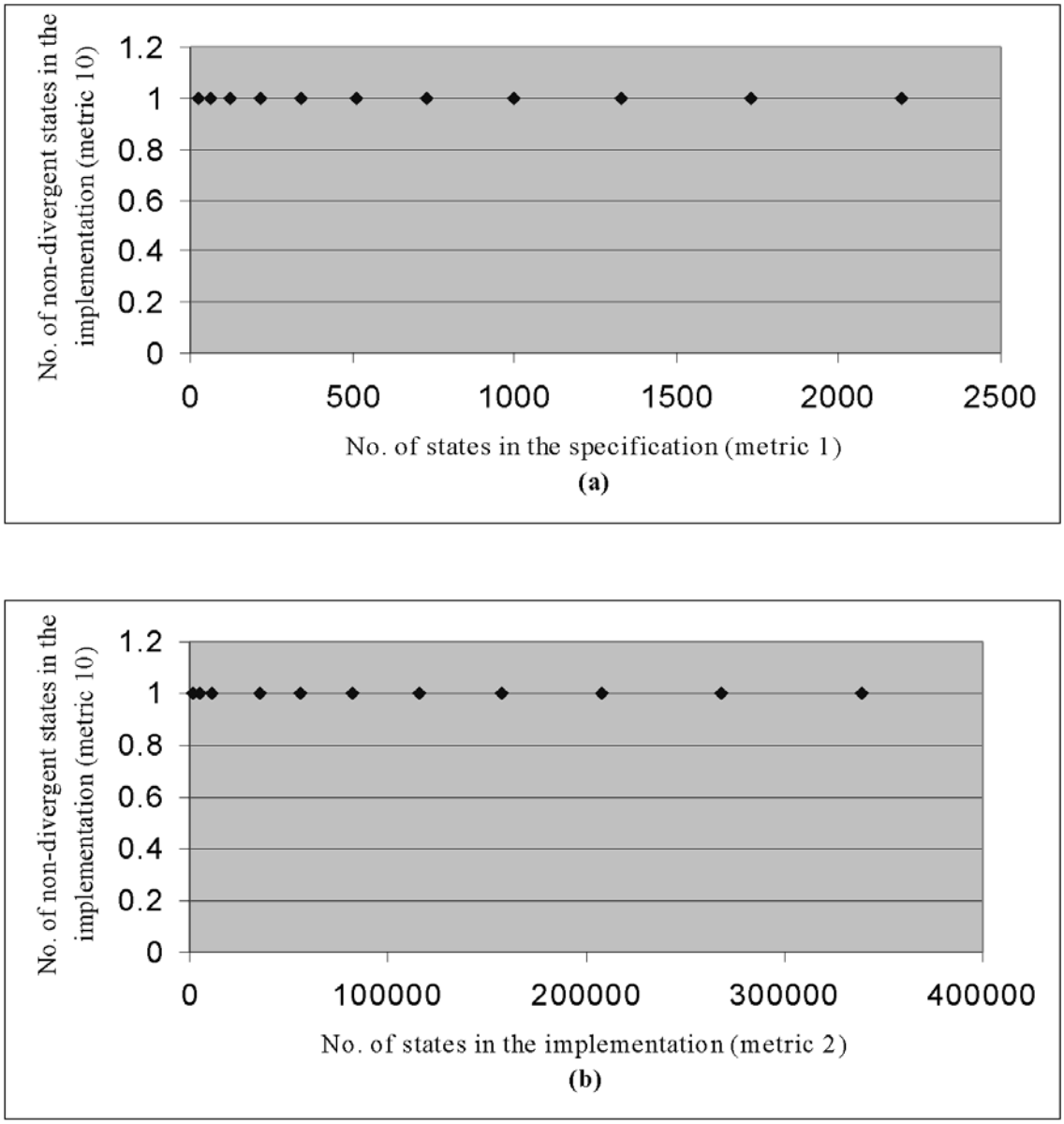
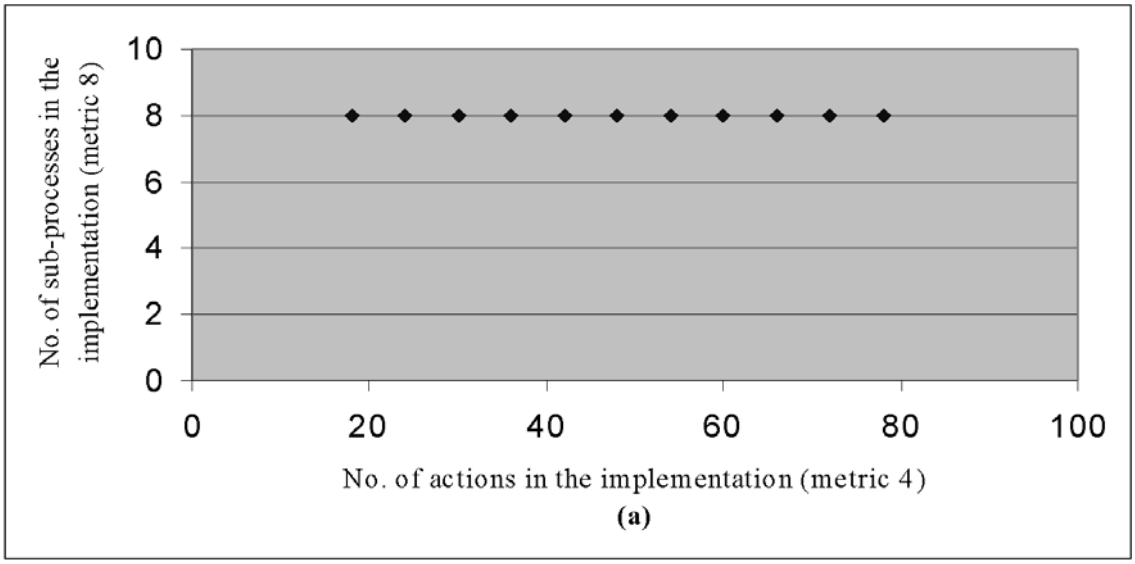


Figure 6.17: Metric 10 versus metric 1 and metric 10 versus metric 2 (LHS)

process looks for a RHS CSP process to communicate with and it fails to do so. LHS generates a large number of implementation states (339300), an indication of the role metric 2 plays in the structure of the overall system.

Interestingly, the number of sub-processes in Figure 6.18(a) remains stable as is the case with Figure 6.17(a). Contrasted with the number of sub-processes generated by both Type II systems, namely the Towers of Hanoi and Dining Philosophers, this clearly shows the lack of variation of System one in terms of the number of sub-processes. Figure 6.18(b) indicates



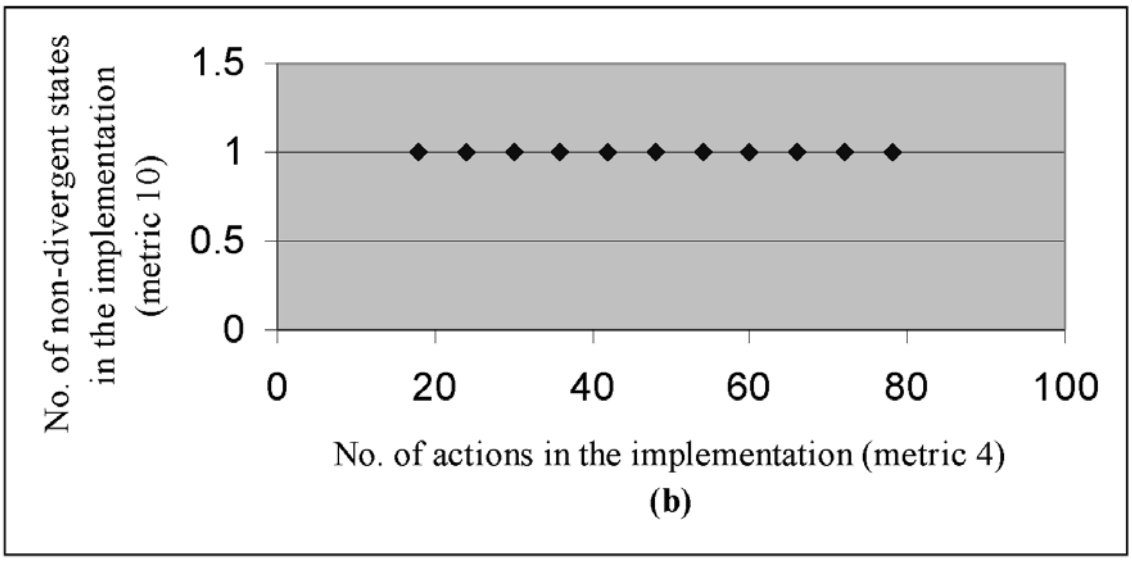


Figure 6.18: Metric 8 versus metric 4 and metric 10 versus metric 4 (LHS)

that whatever behaviour is added to the implementation, the refinement check causes a break down in the expected behaviour of the implementation after just one (non-divergent) state.

The scatter plot in Figure 6.19 shows a linear relationship between the number of states in the implementation and the number of states in the specification. A dashed line (although not visible), representing a *best fit* quadratic curve, overlaps the linear relationship, indicating that when only part of the implementation is used in the refinement check, states in the implementation do not rise as sharply as in the full implementation. Careful inspection of

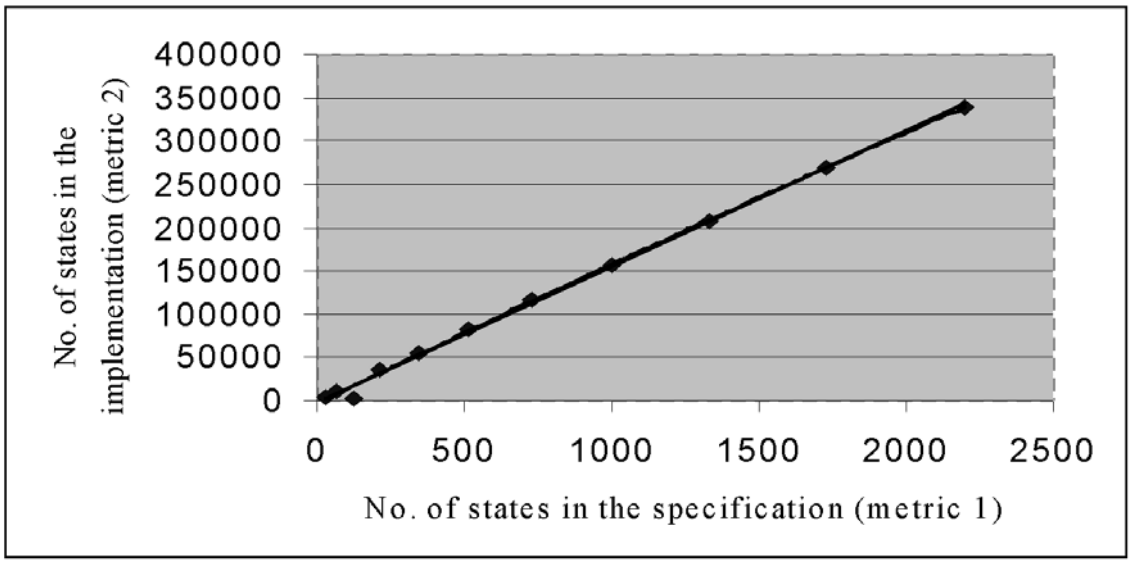


Figure 6.19: Metric 2 versus metric 1 (LHS)

the quadratic curve does show a slight deviation however, suggesting that at higher values on the x-axis, this divergence will increase. The dashed (quadratic) curve in Figure 6.19 has equation y = -0.02x^2 + 158.24x + 1180.1 and the straight line y = 154.09x + 2106; both have Rsv 0.99.

The same number of refinement checks were then carried out for RHS, in effect the mirror image of LHS. Not surprisingly, both LHS and RHS produced identical sets of metrics values and hence have identical scatter plots. We also add that, although the sets of metrics values produced for the two implementations (LHS and RHS) were the same, the two sub-processes did have differing types of action as would be expected from a sending side and a receiving side of a large protocol-based process. For example, the sender always used CSP *send* actions and the receiver CSP *receive* actions using the ? and ! operators, respectively. This is interesting from a metrics perspective, since, in effect, two different processes with different sets of actions can produce the same set of metrics values. The ambiguity lies in the definition of our set of metrics. In order to distinguish one process from another *without* ambiguity in this case, we would require the sets of respective actions (via a set-theoretical

intersection operator) to be compared in order to highlight the differences. It is accepted that this is a weakness in the set of proposed metrics.

We next decompose further the multiplexed buffer system. Just as we have done for `LHS`, so we do the same for `Txs`.

Process two: `Txs`

A set of thirty-three refinement checks were carried out for `Txs` against `Spec` where

```
Txs = ||| i:Tag @ Tx(i)
```

Table 6.8 shows the limits for these refinement checks. The maximum number of implementation states generated by the refinement checks for this sub-process was 161051, compared with 339300 of the process just analysed (`LHS`). The upper limit on the number of states will decrease as the processes are decomposed and become smaller.

Refinement number	Tags	Data values
1 - 10	3	10
11 - 19	4	9
20 - 27	5	8
28 - 33	6	6

Table 6.8: Limits of refinement relationships

The CPU timings in minutes and seconds for the thirty-three refinement checks are given in Table 6.9. It shows clearly the difference between comparing systems under refinement and non-refinement. The mean of 2.49 minutes is far exceeded by that from Table 6.2 (6.54 minutes).

The scatter plots in Figures 6.20 emphasise the similarity of `Txs` with `LHS` in certain characteristics and the differences which arise because `Txs` is a sub-process of the higher-level process `LHS`. For example, the mean number of sub-processes remains relatively static

	min.	max.	median	mean	std. dev.
System one	0.12	37.48	0.17	2.49	6.93

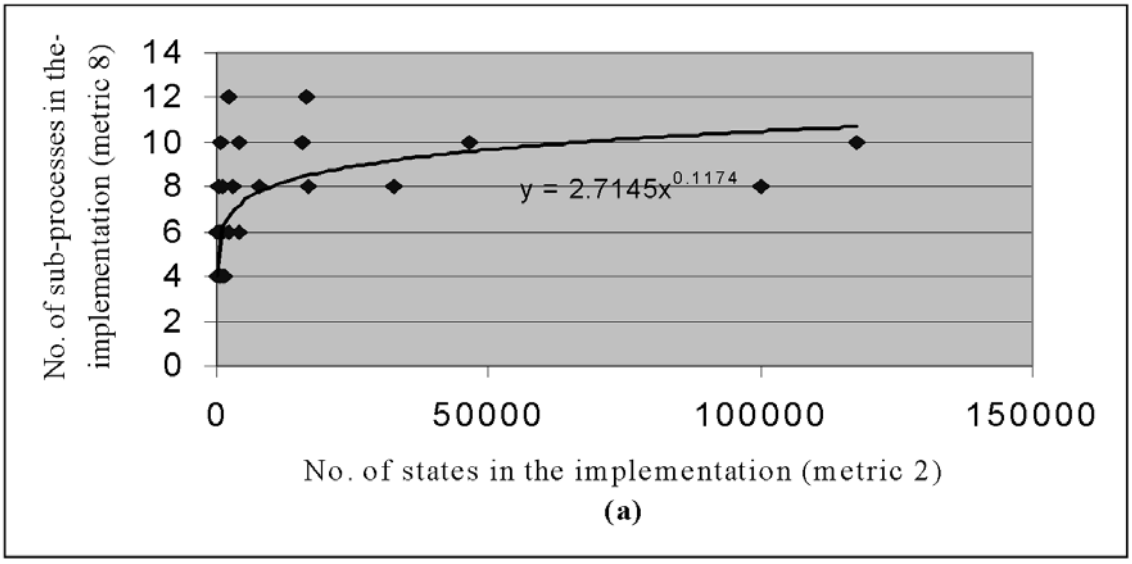
Table 6.9: Summary CPU statistics for System one non-refinement

in `Txs` in common with `LHS`. However, the contribution that `Txs` makes in terms of hidden actions is far less, which is to be expected as the overall system is decomposed. (The outliers in the case of Figure 6.20(a) reflect the wildly fluctuating nature of state expansion for this system; the trend for outliers as more states are generated is still an open problem. An examination of state expansion behaviour generally is still an open research area.)

A set of twenty-nine refinement checks were carried out for the receiver part of the multiplexed buffer `Rxs` defined by

```
Rxs = ||| i:Tag @ Rx(i)
```

against `Spec`. The limits of the refinement checks for `Rxs` were slightly lower than they were for `Txs`, indicating a higher capacity for generating states by `Rxs`. The only other difference between this sub-process and the previous (mirror image) sub-process `Txs` is in the number of non-divergent states in the implementation. Interestingly, the sub-process `Txs` had more non-divergent states than `Rxs` (two against one). This was surprising, since we would have expected the missing behaviour in `Txs` to be identified sooner because it is on the left-side of the process `LHS`. Hence, being encountered first by the FDR model-checker, we would have expected `Txs` to have fewer non-divergent states than `Rxs`. The answer after inspection of the processes and raw data was simple. The `Rxs` has an initial action, which is hidden. The first action of `Txs` is **not** hidden. Hence, the end of non-divergence is signalled earlier with `Rxs`. It is also interesting to note that it is at this level of abstraction that hiding starts to influence the set of non-divergent states in the implementation. Processes at higher



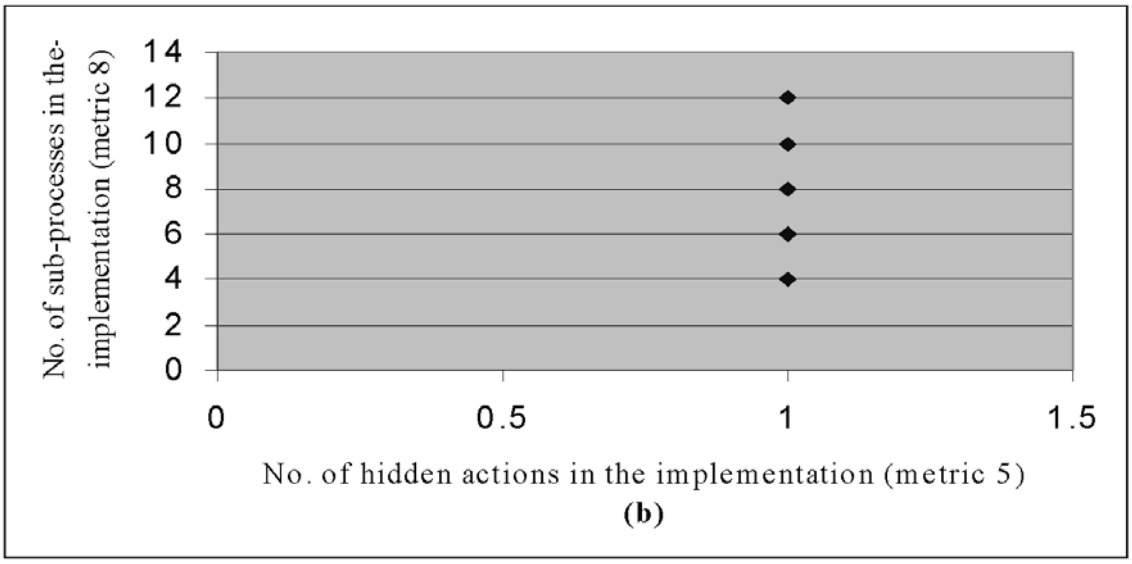


Figure 6.20: Metric 8 versus metric 2 and metric 8 versus metric 5 (`Txs`)

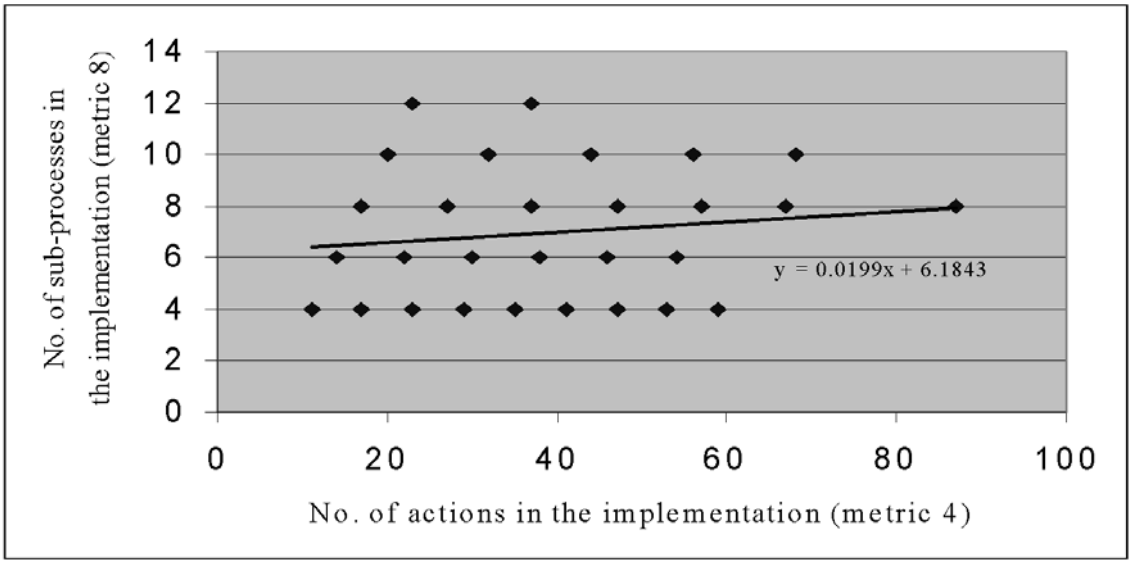


Figure 6.21: Metric 8 versus metric 4 (`Rxs`)

levels of abstraction were not at a low enough level for *hiding* to influence the number of non-divergent states.

The scatter plot in Figure 6.21 is interesting when compared with the corresponding scatter plot for `LHS`, namely Figure 6.18(a), where the number of sub-processes remained static at 8. (We observe that no discernible relationship seems to exist between metric 8 and metric 4 in this case.) Clearly, as the overall system is decomposed and lower levels of abstraction are investigated, the size of the interface grows. In other words, the intensity of communication becomes greater as the layers of encapsulation are removed. We will see when we later analyse the lowest level sub-process of System one, namely `RcvAck`, that there is a limit to this growth which occurs at the lowest level sub-process, i.e., `RcvAck` itself.

Although other scatter plots for `Rxs` were produced, none of them exhibited any other differences between `Rxs` and `Txs`; the most significant result related to this difference is in the number of non-divergent states. Hence, all other scatter plots relating to `Rxs` have been omitted.

We next consider `SndMess` for which thirty-one refinement checks were carried out. Ta-

ble 6.10 shows the limits of these refinement checks.

Refinement number	Tags	Data values
1 - 9	3	9
10 - 18	4	9
19 - 26	5	8
27 - 31	6	5

Table 6.10: Limits of refinement families

Process three: `SndMess`

Process `SndMess` is defined by

```
SndMess = [] i:Tag @ (snd_mess.i ? x -> mess ! i.x -> SndMess)
```

and fits into the overall architecture of the system as follows:

```
LHS = (Txs [|{|snd_mess, rcv_ack|}|](SndMess ||| RcvAck))
      \{|snd_mess, rcv_ack|}
```

The CPU timings for this set of refinement checks are given in Table 6.11.

	min.	max.	median	mean	std. dev.
System one	0.12	5.03	0.17	0.48	0.97

Table 6.11: Summary CPU statistics for System one non-refinement

The single scatter plot produced for `SndMss` is shown in Figure 6.22(a), illustrating the only key difference (i.e., zero sub-processes in the implementation) between `SndMss` and features of the processes at higher levels of abstraction, namely `LHS`, `RHS`, `Txs` and `Rxs`. A corresponding set of thirty-one refinement checks were undertaken for the `RcvMess` sub-process. Scatter plots revealed the same features as for the `SndMess` process (as would be expected from the obvious symmetry of this system).

Process four: `RcvAck`

Thirty-one refinement checks were carried out to evaluate `RcvAck` against `Spec`. For brevity we do not detail the refinement checks.

The `RcvAck` sub-process is defined by

```
RcvAck = ack ? i -> rcv_ack.i -> RcvAck
```

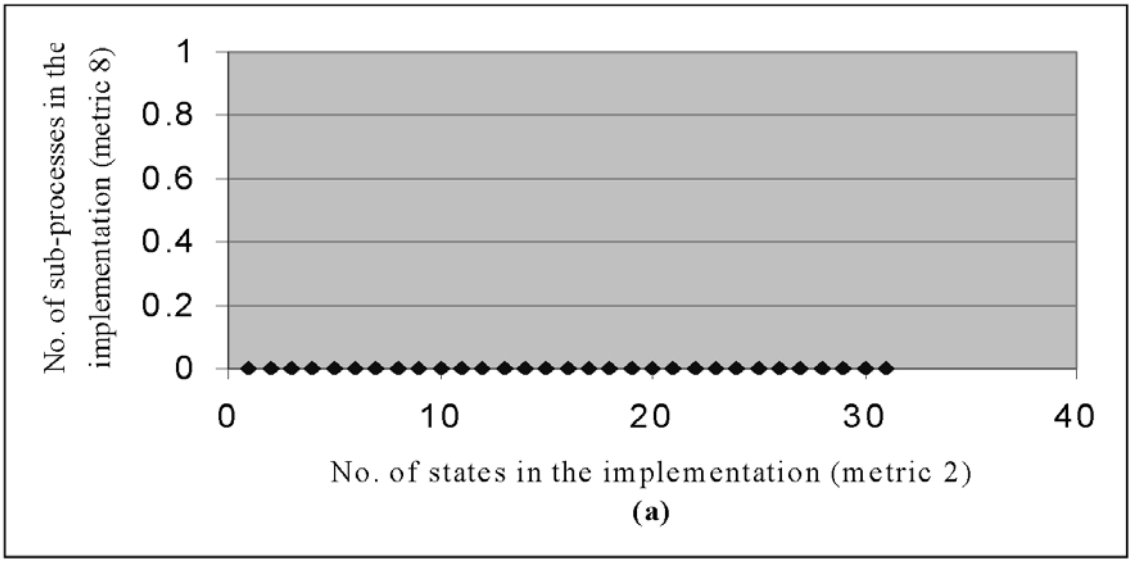
Table 6.12 shows the CPU timings for these thirty-one refinement checks in minutes and seconds.

	min.	max.	median	mean	std. dev.
System one	0.12	7.07	0.16	0.7	1.04

Table 6.12: Summary CPU statistics for System one non-refinement.

In common with `SndMss`, `RcvAck` contained no sub-processes. Figure 6.22(b) shows the relationship between metric 2 and metric 1. The curve in this scatter plot is a much better fit (Rsv 0.83) than the straight line (Rsv 0.34). This difference is remarkable, considering the relative similarity of the Rsv values for the quadratic curve and straight line reported previously in this chapter (see Figure 6.12(b)).

The raw data for this sub-process revealed it to contain no hidden behaviour and no sub-processes. In refinement terms, the bottom level process had therefore been reached (at four levels of nesting). Exactly the same was true of the process `SndAck`. Interestingly, since neither of these two sub-processes had initial hidden behaviour, the number of non-divergent states was identical (i.e., one) for each of the two processes.



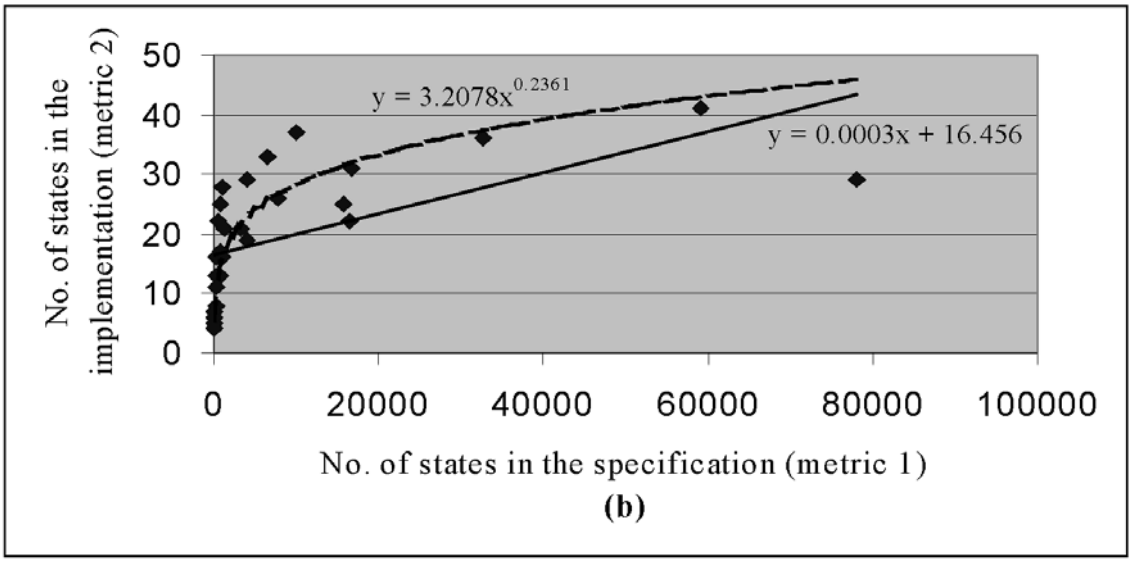


Figure 6.22: Metric 8 versus metric 2 (`SndMss`) and metric 2 versus metric 1 (`RcvAck`)

6.4.2 System one: discussion

For a system such as the multiplexed buffer system, the overall architecture seemed stable in the sense that corresponding sub-processes on either the left side or the right side of the system were fairly similar. What was unexpected was the role that hidden behaviour played in the decomposition, namely, refinement broke down more quickly in sub-processes without hiding (implying that processes which did not have hiding had fewer non-divergent states than those that did). In addition, although the overall architecture is symmetrical in its construction, the metrics did highlight differences between transmitter and receiver.

6.4.3 Alternating bit protocol: non-refinement

Process one: `PUT`

For the alternating bit protocol, the process `PUT` was run against `SPEC`. Eleven refinement checks were carried out. Table 6.13 shows the CPU timings for these eleven refinement checks.

	min.	max.	median	mean	std. dev.
System two	0.11	0.48	0.19	0.22	0.15

Table 6.13: Summary CPU statistics for System two non-refinement

In the implementation there are no hidden actions and no sub-processes in the specification and implementation. Only one non-divergent state in the implementation was encountered before the refinement failed. There are no hidden actions in this process and hence divergence was encountered immediately.

Figure 6.23 shows the scatter plot for states in the implementation versus states in the specification ($Rsv = 0.98$). The shape of the *best fit* curve is different to any of the correspond-

ing scatter plots found for the multiplexed buffer system for the same two metrics (see Figure 6.22, for example). After careful consideration, only one explanation could be put forward. For implementations which incorporate hidden behaviour, the number of states will rise rapidly and continue to do so. However, when the implementation has no hidden behaviour, the number of states in the implementation versus the number of states in the specification will reach a plateau as tentatively indicated by the *best fit* curve in Figure 6.22(b).

Process two: `GET`

A set of eleven refinement checks were also carried out for `GET` versus the `SPEC` process. For `GET`, there were again, no hidden actions. There were no sub-processes in either the specification or the implementation.

The CPU timings for these refinement checks are shown in Table 6.14.

	min.	max.	median	mean	std. dev.
System two	0.11	0.36	0.15	0.19	0.08

Table 6.14: Summary CPU statistics for System two non-refinement

Interestingly, the `GET` process had a static number of implementation states. This meant that the *best fit* curve found for the `GET` process (number of implementation states against number of specification states) was a straight line. The reason for the static number of implementation states can be found in the definitions of the channels `a,b,c` and `d`. The first two channels, used by `PUT`, are defined as `Bool.DATA`; the second two, used by `GET`, are defined to be just `Bool`. This explains why the results for `PUT` and `GET` are different. For the alternating bit protocol, the left and right sides of the process are not as symmetrical as in the case of the multiplexed buffer (`LHS` and `RHS`). In System two, processes `PUT` and `GET`, whilst appearing similar, have distinctly different behaviour.

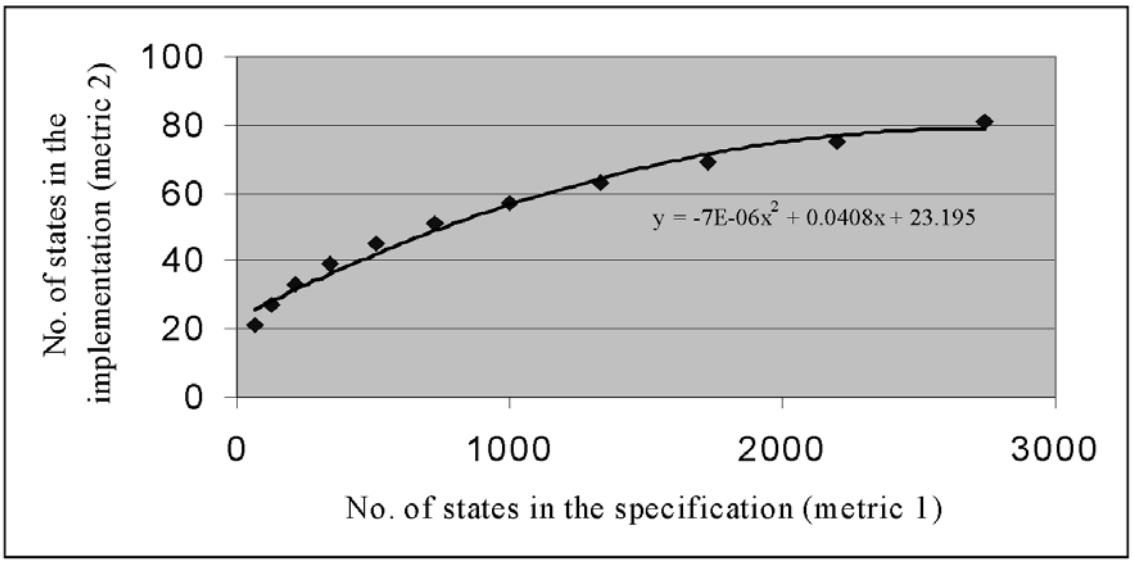


Figure 6.23: Metric 2 versus metric 1 (PUT)

Process three: GET and PUT

A set of eleven refinement checks were carried out for GET and PUT joined together against SPEC. Not surprisingly, the number of states and transitions in the implementation increased from that of GET only and PUT only, as would be expected. The same trend in the relationship between states in the implementation and specification is evident from Figure 6.24 ($Rsv = 0.98$). Since this relationship seemed to be the same for different non-refinements, it was investigated further. It appears, from Figure 6.24, that the *best fit* curve will eventually flatten out; however, using predictive (hill-climbing) techniques to estimate the two values at higher levels for both variables, the curve was found to continue along a similar path and did not in fact flatten out. Had it flattened out, this would have implied that an increase in the number of states in the specification caused no increase in the number of states in the implementation. This would only arise if the two processes shared no actions and therefore no behaviour.

Finally, eleven refinement checks were carried out for RECV against SPEC. These again

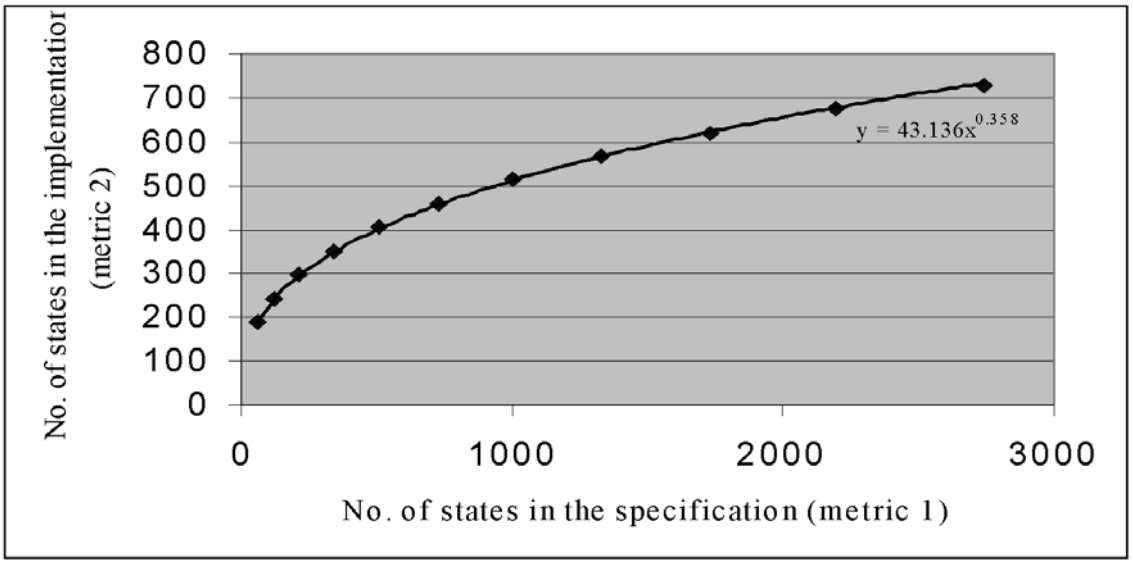


Figure 6.24: Metric 2 versus metric 1 (`GET ||| PUT`)

showed the same relationship between the number of implementation and the number of specification states.

6.5 Conclusions

In this chapter, an empirical evaluation and analysis of the multiplexed buffer and the alternating bit protocol problems has been carried out. Several differences have been identified; quite often these were due to the nature of the application and the resulting architecture. The following main points can be concluded from the analysis.

Firstly, whilst both CSP systems generated a large number of states in the implementation, the multiplexed buffer tended to generate more than the alternating bit-protocol; we attribute this to the architecture of System one, which was able to generate more subprocesses, allowing it to develop a larger set of implementation states.

Secondly, in the context of non-refinement, the number of non-divergent states is determined by the extent of hiding in the behaviour encountered at the beginning of a refinement

check.

Thirdly, in certain cases, different refinement checks can produce the same set of metrics values; further refinement and development of the metrics would be one possibility of resolving this issue. Finally, the relationship between pairs of metrics can vary significantly from the expected.

Although there are no dramatic differences between the two application domains empirically studied in this chapter, there are enough subtle differences to have made the investigation a worthwhile exercise.

In the following chapter, the same empirical treatment is given to the Towers of Hanoi and Dining Philosophers problems. Analysis of these two problems will then allow comparisons between them (as was done in this chapter) as well as between Type I and Type II CSP systems empirically investigated in the thesis.

Chapter 7

Empirical evaluation (Type II CSP Systems)

7.1 Introduction

In this chapter, we focus on the Towers of Hanoi and the Dining Philosophers (Systems three and four); we have called these Type II CSP systems. They are different to the first two systems in terms of their size and construction. This is likely to mean that they will exhibit completely different refinement characteristics; indeed, these systems were deliberately chosen in order to give as broad a range of applications as possible in terms of both the systems' size and their likely behaviour.

The objectives of this chapter are, firstly, to examine whether Systems three and four are similar in the features they both exhibit and, secondly, to examine in more detail the differences between Type I CSP systems and Type II CSP systems. This will then allow us to draw conclusions both across application domains, i.e., those of bit-protocol and problem-solving, and within those two application domains. The analysis will be performed primarily

using the metrics proposed in Chapter 5. Because of the nature of the applications in Type II CSP Systems and the difficulty in extracting meaningful metrics in the context of refinement, the format of this chapter differs from the previous chapter in its structure. Various other problems arose during our empirical investigation of Type II CSP Systems; most of those problems were related to features of the systems themselves. We will describe each of these in turn as the chapter develops. For now, we look at the two systems in isolation.

In Section 7.2, we discuss the motivation for the analysis contained in this chapter. In Section 7.3, we analyse the first of the two Type II CSP systems with appropriate fragments of the CSP code. We also include, where appropriate, histograms and scatter plots to illustrate the major points. In Section 7.4, we analyse the second of the Type II CSP systems. A discussion of the issues raised in this chapter is given in Section 7.5. The main objective of analysing the CSP systems thus far has been to identify traits in the refinement process. Equally important is the relevance and applicability of the metrics themselves in each application domain. One of the major results from the analysis in this chapter was found to be the wide difference in various characteristics between Type I CSP systems and Type II CSP systems. The extent of the differences cast doubt on the scope of the proposed metrics and their applicability for the last two systems. Hence, in Section 7.6, we identify other possible metrics which may give more insight into the characteristics of those last two systems. In Section 7.7, we deliberately introduce faults into the implementation of each of our four systems in order to assess the impact of errors on the values of the metrics. Finally, in Section 7.8, some conclusions about the analysis in this chapter are drawn.

7.2 Motivation

The motivation for the work in this chapter is very similar to that for the previous chapter. Firstly, there are likely to be features of the refinement process and features of CSP processes which have yet to be uncovered through a study of this sort. Secondly, very little is understood about the architecture of systems across different application domains and of different magnitudes. The analysis of a range of systems should highlight trends and characteristics in those systems and provide a basis for further research. In a sense, for a study of the type in this and the previous chapter, there is no such thing as a negative result. Every result, whether confirming or refuting a pre-determined hypothesis informs the next study. Results, in terms of highlighting flaws in the metrics imply that other metrics may be more appropriate. One of the key results from this chapter is that the proposed metrics described in Chapter 5 are inadequate for problem-solving applications. Whilst this was initially disappointing as it highlighted the frailty of our chosen metrics, on reflection, it meant that this knowledge could then be used to propose alternative metrics in Section 7.5.

The analysis described in this chapter therefore tries to capture a little more of the trends and characteristics which emerged from the previous chapter. In common with the structure of the previous chapter, we also look at situations where the implementation refines the specification.

7.3 Towers of Hanoi

A complete lisiting of the code for The Towers of Hanoi can be found in Appendix C. As was the case for Systems one and two, we reiterate the point that the systems were written by experienced CSP developers rather than the author.

The Towers of Hanoi problem is widely known, and is used as a typical example of how a recursive algortihm can be used to solve a problem. An arrangement of differently sized discs on pegs have to be moved from their start position to a final position representing exactly that of the start position, but on a different peg. A restriction on the movement of a disc is that disc a can not be placed on top of another disc b if disc a is larger in size than disc b. In the statement of the initial problem, there are five discs and three pegs (this obviously changed under user control as different refinement checks were undertaken).

```
transparent diamond
n             = 5 -- How many discs
-- Discs are numbered
DISCS         = {1..n}
-- But the pegs are labelled
datatype PEGS = A | B | C
```

For a given peg, we can get a new disc or put the top disc somewhere else. We must also indicate when the peg is full.

```
channel get, put : DISCS
channel full

PEG(s) =
  get?d:allowed(s) -> PEG(<d>^s)
    []
  not null(s) & put!head(s) -> PEG(tail(s))
    []
  length(s) == n & full -> PEG(s)
```

Each `POLE` in the following segment of code represents the completed end status of a peg. Now, given a simple peg we can rename it to form each of the three physical pegs ('poles') of the puzzle. `Move.d.i.j` indicates that disc `d` moves to pole `i` from pole `j`.

```
channel move : DISCS.PEGS.PEGS
channel complete : PEGS

initial(p)   = < 1..n | p == A >

POLE(p) =
  PEG(initial(p))
    [[ full <- complete.p,
       get.d <- move.d.p.i,
       put.d <- move.d.i.p | i <- PEGS, i != p, d <- DISCS ]]

interface(p) = { move.d.p.i, complete.p | d <- DISCS, i <- PEGS }

PUZZLE =
  || p : PEGS @ [ interface(p) ] diamond(POLE(p))



assert PUZZLE \ {| complete.A, complete.B, move |} [F= STOP
```

The puzzle is solved by asserting that `C` cannot become complete. Then the trace that refutes the assertion is the solution.

7.3.1 Refinement details

There are several interesting features of this problem when compared with the examples of the previous chapter; some of these features can be seen immediately by inspection, while others are less intuitive and emerged only after significant investigation.

Firstly, the CSP code of System three is more compact in size than that of the first two. From inspection of the code, there is a large amount of recursion and parameter passing (in common with System two), which aids an elegant and concise description of the problem. Just because the code is compact does not imply, however, that it is any less complex than the corresponding code of the other two problems. In fact, following the analysis of Type II CSP Systems and the difficulty of producing meaningful metrics, one conclusion is that the opposite is the case. Unlike some procedural programming languages, e.g., C, C++ and

Java, complexity in Type II CSP systems seems to be unrelated to its size (in terms of the number of lines of CSP code).

Secondly, an interesting feature of System three relates to the state space generated by the system. The most obvious way to increase the state space of this system is to increase the numbers of pegs and/or discs. However, an immediate effect of doing this is an increase in the amount of recursion the process becomes involved in. Increasing the amount of recursion in a system has an altogether different effect than merely increasing the number of values capable of being passed down a communications channel (as was possible with Systems one and two). The former consumes far more of the available stack space and hence the limit of the stack size imposed by the particular machine is reached more quickly. Type I CSP Systems did use recursion, but nowhere near the extent of System three. Also, in System three, by increasing the scale of the system (i.e., by increasing the number of pegs and/or discs), in most cases[1] what we are doing is effectively adding to the difficulty of solving the problem, not merely to the further generation of states in the process as was the case for Systems one and two). In fact, only a limited number of refinement checks could be completed for System three, resulting in a limited analysis for this system. However, we reiterate that although at first this was disappointing, useful lessons were learnt and incorporated into subsequent analysis.

Thirdly, Type II CSP Systems differ from Type I CSP Systems in that they attempt to find a solution to a problem rather than the generation of large numbers of patterns of bits that can be sent down a communications channel. As such, they are not amenable to decomposition (or the layering of processes) as was found for Type I CSP Systems. This meant that it was difficult to decompose Systems three and four in the same way that was achieved for Systems one and two; for System three, refinement checks were simply

[1] It was observed that in certain cases, increasing the number of discs resulted in a trivial solution, as would be expected, when, for example, there are equal numbers of pegs and discs *or* more pegs than discs.

a check of the implementation against the CSP `STOP` operator (which became, in effect, the specification for this application). `STOP` is a process which denotes termination in an undesirable state (and also one where further progress by the process cannot be made). A good example of an occurrence of `STOP` is that of a robot in a maze reaching the end of a corridor, facing a wall with no way forward and no means of backtracking. A serious implication of running each refinement check against `STOP` is that `STOP` only has one state and zero actions. `STOP` therefore has no state transitions and no traces. Consequently, in the initial refinement checks for System three, the specification has only one state. Despite this being a very restrictive feature, it highlights the limitations that the *type* of problem places on the refinement checks that can be carried out on CSP processes[2]. The emphasis in this problem is on the model-checker finding a sequence of actions which the assertion says does not exist. In short, refinement in the case of System three (and, as we will see, of System four) is not the same as refinement in the case of Systems one and two. The refinement checks that should be carried out are completely different.

Fourthly, for System three, the only variables which could be modified were the number of discs and the number of pegs. There are, however, certain problems associated with extending the state space in this way. A configuration of n pegs and n discs, or any configuration where the number of pegs is greater than the number of discs, will take a relatively short time to solve (since each disc can be placed on its own peg); this simplification of the problem was not a feature of Type I CSP Systems.

Finally, the *scope* of effect that increasing the number of bits had in Systems one and two far exceeded that for Systems three and four. Expressed in another way, addition of datatype

[2]A side-effect of this was that refinement checks to modify the specification, in order to see the effect that that had on the metrics produced, could not be undertaken for this System as they could for Systems one and two (see Chapter 8 for details about modification of the specification).

values in Systems one and two permeated through the whole code of the application; this did not happen for the final two systems. For example, in System one, the lowest level processes are `SndMess` and `RcvAck`. Each of these processes has a series of communications which affect the processes at the next level of abstraction above, i.e., `Txs` and `Rxs`. This in turn affects `LHS` and `RHS`, the top level processes of the system. Such a layering phenomenon was not evident in System three. Closely related to the point just made is that, in effect, although they are larger processes, the way that Systems one and two had been constructed (as a composition of processes) made both of these processes significantly easier to analyse than Systems three and four. In short, Type I CSP Systems one and two are far easier to analyse and comprehend.

In the following section, summary data is provided for System three. A number of the points just raised are reinforced by the values found in Table 7.1, as well as introducing and highlighting other points.

7.3.2 Summary data

Summary metrics for System three in Table 7.1 show values for the metrics of fifteen refinement checks of `PUZZLE` against `STOP` (see Table 7.3). The mean and median values have been rounded up or down where appropriate.

The most striking feature in Table 7.1 is not so much the values of the metrics themselves, as the lack of meaningfulness of the values for the last three metrics; na stands for non-applicability. In the case of metrics 10, 11 and 12, no values can be collected. The reason for this is because of the one-state definition of the specification `STOP`. This explains the absence of values for the three metrics in the table.

An implementation can *only* be compared with a specification as the state space is in-

Metric	min.	max.	median	mean	std. dev.
1 Number of states in the specification	1	1	1	1	0
2 Number of states in the implementation	243	78125	7776	21787	27765.74
3 Number of distinct actions in the specification	0	0	0	0	0
4 Number of distinct actions in the implementation	61	200	116	129	44.44
5 Number of hidden actions in the implementation	7	16	10	10	2.95
6 Number of visible actions in the implementation	0	0	0	0	0
7 Number of sub-processes in the specification	1	1	1	1	0
8 Number of sub-processes in the implementation	44	44	44	44	0
9 Number of additional actions in the implementation	61	200	116	129	44.44
10 Number of non-divergent states in the implementation	na	na	na	na	na
11 Number of transitions in the implementation	na	na	na	na	na
12 Number of harmless partially divergent states	na	na	na	na	na

Table 7.1: Summary metrics for System three

creased, if it is possible to increase the specification state space as well. Central to the analysis in the previous chapter was the divergence in specification and implementation metrics as the state space was increased; this was not possible for System three. This scenario may therefore represent a flaw in the metrics themselves, and cast doubt on their ability to capture features of applications other than for bit-protocol problems. We will examine the appropriateness of our metrics in Section 7.6 and suggest other metrics which may have been more appropriate. For now, however, we continue with our analysis of System three.

The number of visible actions in the implementation is zero. This is because the implementation contains largely hidden behaviour. This can be seen clearly in the definition of the assertion

```
ASSERT PUZZLE \ {| complete.A, complete.B, complete.C, move |}
```

In other words, all behaviour related to `move` and `complete` is hidden. These actions represent the majority of the behaviour for this application. Compare this configuration with the two protocol-type problems where a fairly even balance between hidden and visible behaviour was identified (in System one, for example, only channels `mess` and `ack` were hidden whilst `left` and `right` were visible; in System two, the channels `a,b,c,d` were all hidden

whilst `left` and `right` were visible). The differences between System three and Type I CSP Systems become evident in this sense. Generally speaking, Type I CSP Systems conform to the principles of abstraction and the black-box approach. This does not seem to be the case for System three; this is attributed to the nature of the problem.

Also noticeable in Table 7.1 is the relatively large number of actions in the implementation (metric 4, maximum 200) and the relatively small number of states generated in the implementation (metric 2, maximum 78125), when compared with the previous two systems (see Tables 6.1 and 6.4). Compare the maximum number of actions and states in the implementation for the multiplexed buffer system (161 and 364440, respectively) and for the alternating bit protocol (18 and 452112, respectively), and the difference becomes apparent. The reason for this is due to the large number of combinations of `PEG` and `DISC` which evolve as the state space is increased. The set of actions that System three was capable of engaging in represented every possible combination of movement of a peg to a disc. As the number of pegs and/or discs grew, the possible movement of discs across `PEG` grew at an alarming rate.

It can be seen from Table 7.1 that the number of sub-processes in the implementation remains static at 44 as the state space increases. One explanation for this might be that the architecture of System three remains fairly static; it is the possible behaviour around that architecture which changes.

The minimum, maximum, median and mean CPU timings are provided in Table 7.2.

	min.	max.	median	mean	std. dev.
System three	0.17	9.38	0.56	2.14	3.23

Table 7.2: Summary of CPU statistics for System three

Interestingly, the CPU timings for System three are significantly less than those for Systems one and two. The mean time for a refinement check for the multiplexed buffer was 6.54

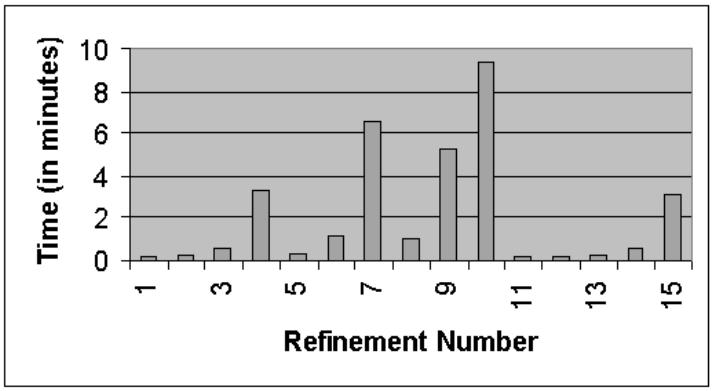


Figure 7.1: Histogram of CPU timings

minutes (See Table 6.2). For the alternating bit protocol the mean was 8.91 (See Table 6.3). There are a number of explanations for these relatively low timing figures. System three is heavily recursive in nature. The objective of the problem is to arrive at a solution (via the assertion) rather than a declaration of refinement (as found in Systems one and two). We also need to take into account the fact that the specification only contains one state, meaning that the majority of the processing time is not shared between analysing the specification *and* the implementation (as for the previous two systems); only the implementation has to be analysed. Allied to this is the ability of the model-checker to converge on a solution quickly (CSP lends itself readily to extensive use of recursion), and the reasons for relatively low CPU timings start to become apparent.

For both systems in the previous chapter, a limit on the state space of the problem appeared to exist. For the multiplexed buffer problem, the upper limit was sixty-two minutes (for the five `Tags` and three `Data` values configuration). For the alternating bit protocol, the limit was just over sixty-four minutes. Inspection of the raw data for the corresponding limit in the case of System three revealed the timing to be for a configuration of six `PEGS` and seven `DISCS`. The histogram in Figure 7.1 represents the timings for the different combinations of `DISCS` and `PEGS` for which refinement checks were undertaken. The lowest timing was found

for the three PEGS, five DISCS configuration which took 14 seconds to complete. Figure 7.1 shows that for ten of the fifteen refinement checks, the problem was solved in a relatively short time.

For clarity, we give below Table 7.3 corresponding to Table 6.3 in the previous chapter.

Refinement number	PEGS	DISCS (range)
1 - 5	3	5 - 9
6 - 9	4	5 - 8
10 - 12	5	5 - 7
13 - 15	6	5 - 7

Table 7.3: Limits of refinement families

7.3.3 Refinement scatter plots

Due to the sparsity of metrics values for System three, due primarily to the specification comprising only one state and one action, there was, consequently, a severe limit on the number of meaningful scatter plots that could be produced. In the previous chapter, certain scatter plots were omitted from the analysis because the relationships between the metrics were either obvious or had no intuitive meaning and therefore did not warrant investigation. The same applies in the case of System three. After taking the obvious relationships for System three into account, and also relationships where the metrics remain constant (for example, in the case of the number of sub-processes), there were only a few remaining relationships worth investigating. Later in the chapter (and to a larger extent in the conclusions and evaluation chapter), we take time to reflect on the features of the four systems which prevented or hampered further analysis.

One relationship worth investigating for system three, however, because of the similarity with results found for Type I CSP Systems, is shown in the histogram of Figure 7.2, between actions and sub-processes in the implementation (metric 4 and metric 8).

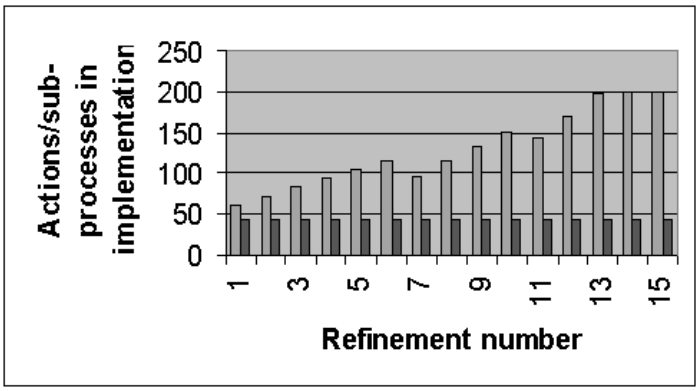


Figure 7.2: Actions and sub-processes in the implementation against refinement checks. (Light bars represent actions; dark bars represent sub-processes.)

When Figure 7.2 is compared with Figure 6.9 of System one, it is clear that in both systems there is little variation in the number of sub-processes as the state space is increased. On the other hand, the rate at which new actions are introduced into the implementation increases gradually, in general, for System three and fluctuates for System two.

7.3.4 Refinement of System three: conclusions

There is clearly a big difference between the Towers of Hanoi problem and the two protocol-type problems of the previous chapter. These differences cover a multitude of features, from the size of the system in terms of its physical layout on a page, to the way it is written, whether that be heavily recursively and/or based on extensive parameter passing.

The first conclusion from looking at System three is therefore that the choice of metrics is crucial to obtaining the most information about a process. Clearly, the twelve metrics proposed in chapter five are inadequate for the Towers of Hanoi problem, and alternative metrics need to be investigated.

A further observation relates to the CPU timings for System three. Close inspection of the data revealed that, in one case, the CPU timing actually went down upon addition of an extra disc. This would imply that addition of the extra disc caused the solution to become

trivial. This was not a feature of Systems one and two.

7.4 System four: dining philosophers

The Dining Philosophers problem is similar to the Towers of Hanoi problem in its dependence on recursion. However, there is one essential difference, namely, that the possibility exists for deadlock to occur at some point in the process of solving the problem.

The full code for System four is contained in Appendix D. The definition of the Dining Philosophers problem is as follows:

> There are five philosophers sitting round a table and ten forks. Each philosopher does two things: eat and think, these activities being interleaved. In other words, a philosopher will eat, think for a while and then eat again. However, a philosopher can only eat if he/she has control of the fork to the left and the fork to the right of them (let us assume they are eating spaghetti). The problem is one of avoiding a situation where every philosopher has a single fork and is unable to obtain a second fork, hence a situation where nobody can eat (thus causing starvation, a form of deadlock).

Looking at the CSP code for this problem, it can be seen that the Dining Philosophers problem is similar in its compactness to System three. The only effective way of increasing the state space is by increasing the number of philosophers. However, similar problems arose when trying to increase the size of the state space of System four as for System three. The number of philosophers could only be raised to nine before the number of sub-processes as a result of recursive calls became unmanageable and exceeded the size of the virtual memory of the machine (rhea.dcs.bbk.ac.uk) on which the experiments were run.

```
datatype Move = Up | Down
channel left, right : Move

FORK = left.Up -> left.Down -> FORK [] right.Up -> right.Down -> FORK
PHIL = left.Up -> right.Up -> left.Down -> right.Down -> PHIL

LPHILS(n) =
  let
    L(0) =
      FORK [right<->left] PHIL
    L(n) =
     let
        HALF = LPHILS(n-1)
     within HALF [right<->left] HALF
     transparent normal
  within normal(L(n) [[ ]])

RPHILS(n) =
  LPHILS(n) [[ left <- right, right <- left ]]

PHILS(n) =
  LPHILS(n-1) [| {| left, right |} |] RPHILS(n-1)

-- PHILS(n) represents a network of 2^n philosophers

X = PHILS(4)

assert X [FD= X
```

Summary metrics for System four are given in Table 7.4. Again, the mean and median values have been rounded up or down where appropriate.

Metric	min.	max.	median	mean	std. dev.
1 Number of states in the specification	1	3	1	2	1
2 Number of states in the implementation	1	3	1	2	1
3 Number of distinct actions in the specification	6	6	6	6	0
4 Number of distinct actions in the implementation	6	6	6	6	0
5 Number of hidden actions in the implementation	5	5	5	5	0
6 Number of visible actions in the implementation	1	1	1	1	0
7 Number of sub-processes in the specification	9	1533	777	774	876.43
8 Number of sub-processes in the implementation	9	1533	777	774	876.43
9 Number of additional actions in the implementation	0	0	0	0	0
10 Number of non-divergent states in the implementation	na	na	na	na	na
11 Number of transitions in the implementation	na	na	na	na	na
12 Number of harmless partially divergent states	na	na	na	na	na

Table 7.4: Summary metrics for System four

The first point to note from Table 7.4 is that the number of states in the implementation is the same as the number of states in the specification. As in the previous example, the assertion tested was

```
assert X [FD= X
```

In other words, the specification is set to be equivalent to the implementation; this was done deliberately and through necessity; the aim of the implementation is to find a deadlock-free solution, rather than decide on refinement or not. Hence, the assertion tested in this case differs from the original description of the Dining Philosophers problem, where the asertion is that the solution is free from deadlock. The important point is that, because of the way in which the metrics collection code had been written, i.e., tailored primarily to protocol type problems, none of the metrics could have been extracted if the assertion had been one of deadlock freedom. Another way of expressing this is to say that the Dining Philosophers problem is not amenable to the same sort of refinement check as the two protocol-type problems. The same problem therefore arises with System four as did for System three. The set of proposed metrics is inadequate for capturing features of systems typified by Systems three and four.

The second point worth noting from Table 7.4 is the maximum value of the number of sub-processes in the implementation (metric 8). For the four refinement checks carried out, the number of sub-processes rose from five sub-processes for one philosopher, to nine for two philosophers. The number of sub-processes then remained unchanged at 1533 sub-processes for three and four philosophers. It would have been interesting to have increased the number of philosophers further in order to view the effect this had on the number of sub-processes. However, this was not a practical consideration as the stack space of the machine (rhea) was exhausted.

Finally, the number of actions in the implementation (metric 4) remains static despite the number of philosophers being increased. The number of actions in the implementation for this application is *always* fixed, since there are only a specific set of moves that can be made. This represents another difference between System four and Systems one, two and, in this case, System three. The list of actions for System four are:

```
tau
right.Up
right.Down
left.Up
left.Down
tick
```

However, in common with the Towers of Hanoi problem, the majority of actions in the implementation (apart from the `tau` action) are hidden actions (see metric 5).

Table 7.5 shows the statistics for the CPU timings for System four. The minimum, maximum, median and mean values are all at 0.09 seconds. The similarity of these values is difficult to interpret. One would imagine that, in view of the extensive use of recursion in System four, the CPU timings would rise rapidly as the number of philosophers was increased; this was obviously not the case. One explanation for this might be that the problem is essentially one of avoiding a deadlock situation, i.e., a situation where none of the philosophers can pick up a second fork. Hence, as soon as it had been established that the system did or did not deadlock, the process terminated (irrespective of the assertion being tested). Addition of extra philosophers had no impact on the time it took to reach the conclusion regarding freedom from deadlock.

	min.	max.	median	mean	std. dev.
System 1	0.09	0.09	0.09	0.09	0

Table 7.5: Summary CPU statistics for System four

7.5 Discussion

The two problems analysed in this chapter have certain similarities. One major similarity between Systems three and four was in the small number of states in the implementation generated at the upper limit of the problem, vis-*à*-vis Systems one and two. In the case of System four, this was a combination of: firstly, the memory restrictions imposed by the machine generating those states, and, more specifically, the stack space available; secondly, the number of actions in the implementation (metric 4) which remained static as the state space increased. In the case of System three, the number of actions in the implementation is higher than any of the other three systems. However, it is the limit imposed on expansion of the state space through the influence of recursion which limits the state space in this case. Hence, the difference is that for Systems one and two, the upper limit in the size of the state space was restricted by the amount of virtual memory. In Systems three and four, it was the size of the stack which impacted the size of the state space.

The results in this chapter also suggest that for a particular programming language, there are no universal metrics which capture features of every program derived from the language. This comes as no surprise; the same is true of the object-oriented paradigm. Some metrics will always be countable in the OO sense, for example, the number of classes, methods and attributes. Other metrics may yield no meaningful values; evidence suggests [CNM00] that aggregation in OO systems is used very sparingly. Ultimately, it is a question of tailoring the choice of metrics to the particular application domain being analysed. In addition, the

nature of the application domain, in this case a problem-solving environment, will determine how well an implementation can be compared with its specification.

The key feature of this discussion is the inadequacy of the metrics proposed in Chapter 5. In the following section, using the experience of collecting metrics from Systems three and four (and the experiences from Systems one and two), a number of alternative metrics are proposed which may provide a greater insight into the characteristics of the last two systems.

7.5.1 Alternative metrics

To propose metrics which are more meaningful for the Type II CSP systems, we must consider the features which were missing from Systems three and four, but were present in Systems one and two (and vice versa). Expressed in another way,

> Why were the metrics more appropriate for bit-protocol type applications than for problem-solving applications?

One suggestion why the metrics in Table 5.1 were appropriate for Type I CSP Systems but not for the last two systems is that when developing those metrics, only features of a bit-protocol type problem were considered. Refinement was the key issue from the beginning and the metrics were designed to capture essential differences between specification and its implementation.

In the ensuing sections, we consider features pertaining to all CSP systems and their potential for capturing features of Type II CSP systems more accurately than the original proposed set of metrics. Our consideration is heavily influenced by the results of our empirical analysis.

Channel communications

The number of actions and states in a CSP process largely reflect the number of communication patterns in the application being considered. This feature only seems to manifest itself in applications where the state space is not limited by the stack space of the machine. In other words, it is worth considering how many of the operators in each of the four systems were either an input from or output to a channel. By inspection, the number of communications are fourteen for System one and thirteen for System two. This compares with two for System three and zero for System four. There is clearly a difference in the volume of communication and therefore patterns of communication between the four systems.

Recursion

Another feature which differed between the two types of application was the amount of recursion. One metric which may be useful to know in order to assess the potential for state space expansion is the number of static recursive calls in the CSP implementation. Examination of the implementation of System one revealed it to contain eight static recursive calls (the specification contains one recursive call). The implementation of System two contained twelve static calls. The implementation of System three contained three static recursive calls and System four contained three. This metric, however, is clearly not as applicable as others, since what we are really interested in is the dynamic (run-time) extent of recursion, which this metric (based on our experience of analysing CSP processes) does not give. This run-time metric is a potential new metric for Type II CSP systems and requires further research.

Renaming

Another feature which seemed to distinguish the two types of domain was the number of renaming operations. In Systems one and two, there are zero renaming operations. In Systems three and four there are seven and two renaming operations, respectively.

Interleaving and non-determinism

System one contained four interleaving operations (one of which was contained in the specification). This reflects all the possible combinations of transfers of data between sub-processes. System two has three non-deterministic operators, because the choice of channel a value should be received from is not deterministic. Type II CSP Systems three and four have neither of these CSP features. Whilst highlighting the differences between the four systems, the number of interleaving and non-deterministic operators are two candidate metrics for extension of the original twelve metrics in Systems one and two (but not for Systems three and four).

Retained metrics

There are some metrics of the initial proposed set which do accurately reflect features of each of the four CSP systems. These include:

- The number of states in the implementation.
- The number of visible actions in the implementation.
- The number of hidden actions in the implementation.

To the above three metrics we add the following two metrics, which appear to be common in Systems three and four:

- The number of communication-based structures in the implementation.
- The number of renaming operations in the implementation.

Finally, as a dynamic measure of recursion (rather than a static one, which does not reflect *run-time* recursive behaviour) we propose the new metric:

- The number of sub-processes in the implementation.

This gives a set of six metrics for capturing features of problem-solving applications. These six metrics represent a mixture of dynamic and static measures. This is interesting as it suggests that a set of applicable metrics should contain both static and dynamic metrics.

The question then has to be asked: do these metrics give more insight into the process of refinement? The answer to this question is that the last two CSP systems do not tend to engage in significant levels of refinement as evidenced by the empirical analysis. The assertions tested in these two systems are tests of refinement against trivial processes. In other words, testing any features of refinement in systems such as those of Systems three and four is likely to reveal very little. Although in a sense this is disappointing, it is a worthwhile result to be aware of, since it suggests that refinement is not always used in the same sense for each type of application domain.

For the CSP processes analysed in this and the previous chapter, it would seem that there is a process *scale*, on one end of which are bit-oriented CSP processes and at the other end are problem-based CSP processes. The features of protocol-based applications are primarily (as would be expected) a dependence on communication events, hiding and interleaving. The features of problem-based applications are a limited amount of communication through channels, a high dependence on renaming (relabelling) to fufil the functionality of the problem, and significant amounts of run-time (dynamic) recursion. To justify the choice

of metrics and go some way to supporting the claim of a scale along which CSP processes of different application types can be found, a final CSP system was analysed and the above six metrics were collected. We hasten to add that analysis of this new CSP system was not on the scale of the previous Systems (one to four).

7.5.2 Railway crossing system

The railway crossing example is a CSP process which models the operations of trains on railway tracks, the controls that ensure gates operating access to the railway open and shut at the right time, and the timing constraints that apply when a train is between certain sensors. The logic of the problem is quite complex, and there is also a need for strict communication rules between the competing elements of the system, i.e., the trains, signals, sensors, etc.

The railway crossing problem (see Appendix E) was chosen because it represents neither a pure communication-based application, nor a purely problem-based application. Initial analysis revealed it to contain both non-deterministic and interleaving operations. It also contained an average level of communication-based operators, i.e., less than the two Type I CSP systems yet more than the two Type II CSP systems; it also contained renaming. The static metrics for this CSP process were:

- The number of visible actions in the implementation (3).
- The number of hidden actions in the implementation (47).
- The number of communication-based structures (59).
- The number of renaming operations (1).

For this system, two refinement checks were carried out [3].

For the initial configuration (of 10 track segments), there were 23171 states in the implementation, and 49 sub-processes in the implementation. This application contained features found in both the bit-protocol systems (with large numbers of communication-based structures) and the problem-solving systems (with large amounts of hidden behaviour and sub-processes). Increasing the track segments to 12, resulted in the following set of metrics:

- The number of visible actions in the implementation (3).
- The number of hidden actions in the implementation (55).
- The number of communication based structures (59).
- The number of renaming operations (1).

For the changed configuration, there were 34427 states in the implementation, and 58 sub-processes in the implementation. It thus became apparent very quickly that for this CSP system, some features resembled those of Type I CSP systems and others those of Type II CSP systems. In terms of the characteristics of this railway crossing problem, evidence suggests that it lies somewhere between Type I and Type II, supporting the claim that bit-protocol and problem-solving applications are, in effect, at opposite ends of the CSP process spectrum.

7.6 Fault-based analysis

A certain number of refinement checks were undertaken for each of the four systems analysed to view the effect of seeded faults on the metrics values. The main objective was to determine

[3] At first, an attempt was made to increase the state space by increasing the number of trains from its current initial value of two. However, this proved to be an unmanageable problem for the model-checker.

how the metrics values change as a result of faulty behaviour being introduced. In particular, to contrast the metrics values obtained in the context of faults with those obtained from the fault-free analysis carried out earlier; changes in the relationship between specification and its implementation (for each of the four systems) are bound to be reflected in the metrics values thereby obtained. Subjecting the four CSP systems to different types of analysis is also likely to increase our understanding of the behaviour of those four systems.

7.6.1 System one: multiplexed buffer

The choice of which faults to introduce was determined in the case of System one by the existence of a faulty system (called appropriately `FaultySystem`), written by the original developer, as an example of how the refinement process can become faulty. The first refinment check carried out was therefore `Spec` versus `FaultySystem`, for which twenty-two refinement checks were carried out, for varying numbers of `Tags` and `Data`. Table 7.6 indicates the refinement limits for these twenty-two checks.

Refinement checks	Tags	Data values
1 - 10	3	10
11 - 16	4	6
17 - 19	5	3
20 -21 37	6	2
22	7	1

Table 7.6: Limits of refinement families

The faulty version of the implementation centres around the ability (or not) of the subprocess `FaultyRx(i)` to send an acknowledgement after outputting a value on the `right` channel, namely,

```
Rx(i) = rcv_mess.i ? x -> right.i ! x -> snd_ack.i -> Rx(i)
```

```
FaultyRx(i) = rcv_mess.i ? x -> right.i ! x ->(FaultyRx(i)
                                                |~| snd_ack.i -> FaultyRx(i))

FaultyRxs = Rx(t1) ||| Rx(t2) ||| FaultyRx(t3)

FaultyRHS = (FaultyRxs [|{|rcv_mess, snd_ack|}|]
                        (RcvMess ||| SndAck))\{|rcv_mess, snd_ack|}

FaultySystem = (LHS [|{|mess, ack|}|] FaultyRHS)\{|mess, ack|}
```

Table 7.7 contains the summary metrics for the twenty-two refinement checks. The most

Metric	min.	max.	median	mean	std. dev
1 Number of states in the specification	8	2401	250	511	537.70
2 Number of states in the implementation	544	239784	27062	54130	70710.50
3 Number of distinct actions in the specification	8	62	26	30	15.79
4 Number of distinct actions in the implementation	26	161	73	81	40.30
5 Number of hidden actions in the implementation	6	33	17	18	8.20
6 Number of visible actions in the implementation	20	128	56	63	32.11
7 Number of sub-processes in the specification	4	12	5	6	2.39
8 Number of sub-processes in the implementation	21	29	23	23	4.78
9 Number of additional actions in the implementation	18	99	47	51	24.61
10 Number of non-divergent states in the implementation	18	9827	168	1519	106294.80
11 Number of transitions in the implementation	1036	406116	47153	98602	154166.70
12 Number of harmless partially divergent states	518	203058	23577	49299	68755.06

Table 7.7: Summary metrics for fault-based analysis (System one)

noticeable feature in Table 7.7 is the relatively small number of non-divergent states (metric 10) when compared with the number of states in the implementation (metric 2). These two metrics values are plotted in the histogram of Figure 7.3 against the refinement number and show clearly the influence of the fault on the number of non-divergent states (represented by the darker bars).

The other metrics values are comparable to the initial analysis of Chapter 6. Thus, the only metric which reflects the influence of the fault is the number of non-divergent states. This could be seen as another criticism of the set of proposed metrics since only one of the metrics gives worthwhile information about the injected fault.

The CPU timings for the fault-based analysis were comparable to those for the analysis

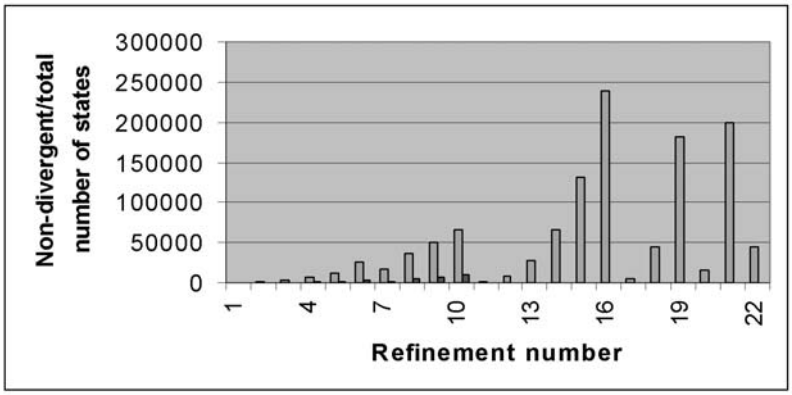


Figure 7.3: Fault-based analysis: System one

contained in Chapter 6.

7.6.2 System two: alternating bit protocol

For System two, thirteen refinement checks were carried out (the number of `DATA` values in the body of the code was raised from 3 to 15 to achieve this). Unlike for System one, no faulty system exists in the technical literature. The choice of fault to investigate in this case was strongly influenced by the type of fault investigated for System one. The advantage of this approach is that we can then compare the resulting metrics between Systems one and two.

The following section of code is taken from System two and reflects the code used in the analysis of the system in the previous chapter.

```
R(bit) =
       b?tag?data -> (if tag==bit then right!data ->
                                    R(not bit) else R(bit))
       []
       c!not bit -> R(bit)
within R(false)
```

This is followed by the code with a seeded fault, similar to that used for System one.

```
R(bit) =
      b?tag?data -> (if tag==bit then (right!data ->
                                       R(not bit) |~|
                                       R(not bit)) else R(bit))
      []
      c!not bit -> R(bit)
within R(false)
```

Table 7.8 contains the summary metrics for the thirteen refinement checks. The table only includes the three most relevant metrics for this analysis; the previous analysis for System one revealed that the majority of the set of metrics were similar to those found for the initial analysis in Chapter 6. Hence, for this type of analysis of System two, we accept that only certain metrics are relevant.

Metric	min.	max.	median	mean
Number of states in the specification	64	3375	729	1107
Number of states in the implementation	900	12132	5220	5724
Number of non-divergent states in the implementation	687	9479	4037	4444

Table 7.8: Summary metrics for fault-based analysis (System two)

Another conclusion from the research in this thesis would be that for fault analysis, only a certain subset of the proposed metrics is applicable.

The histogram of non-divergent states and the number of states in the implementation against refinement number is shown in Figure 7.4 (the darker bars represent the number of non-divergent states). Interestingly, and in contrast to System one, the number of non-divergent states rises proportionately with the number of states. The relatively larger number of non-divergent states in System two is due to the fault occurring at a far later stage in the receiving end of the system.

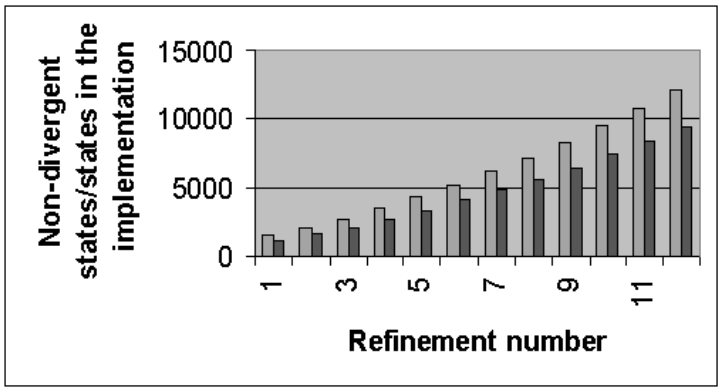


Figure 7.4: Fault-based analysis: System two

7.6.3 System three: towers of Hanoi

For System three, nine refinement checks were carried out, with the number of `PEGS` held at 3 and the range of `DISCS` ranging from 5 to 13. The injected fault was embedded in the code as follows:

```
R(bit) =
POLE(p) =
  PEG(initial(p))
    [[ full <- complete.p,
       get.d <- move.d.i.p,
       put.d <- move.d.i.p | i <- PEGS, i != p, d <- DISCS ]]

assert STOP [FD= PUZZLE \ {| complete.A, complete.B, move |}
```

In this segment of code, the `move.d.i.p` occurs twice. The first of these moves should have been `move.d.p.i`. Running the refinement revealed that the implemention refused to engage in an event which should have been allowed after just one state. This meant that for each of the refinement checks, there was only ever one non-divergent state.

Table 7.9 contains the summary metrics for the nine refinement checks undertaken for this process. From this table, it is even more evident that the original set of twelve metrics is inappropriate for fault-based analysis.

Metric	min.	max.	median	mean
Number of states in the specification	1	1	1	1
Number of states in the implementation	1	1	1	1
Number of non-divergent states in the implementation	1	1	1	1

Table 7.9: Summary metrics for fault-based analysis (System three)

A further set of four refinement checks (for the 3 `PEGS`, five, six, seven and eight `DISC` combinations) were carried out with the following fault seeded modifications

```
interface(p) = { move.d.i.p, move.d.p.i, complete.p | d <- DISCS, i <- PEGS }

assert PUZZLE \ {| complete.A, complete.B, move |} [F= STOP
```

In this segment of code, `move.d.i.p` occurs in front of the `move.d.p.i`. There is an added move and that move is erroneous.

Table 7.10 contains the summary metrics for these four refinement checks undertaken for this scenario. The histogram of states and non-divergent states in the implementation

Metric	min.	max.	median	mean
Number of states in the specification	1	1	1	1
Number of states in the implementation	243	6561	1458	2430
Number of non-divergent states in the implementation	241	6313	1438	2358

Table 7.10: Summary metrics for fault-based analysis (System three)

against refinement number is shown in Figure 7.5 (the darker bars represent the number of non-divergent states). Interestingly, the gap between the number of implementation states and non-divergent states remains constant until refinement number three, suggesting that the role of the interface does not change between five and seven discs (the solution is trivial with six discs). Carrying out refinement checks beyond eight discs led to a segmentation fault in the machine. However, from this small analysis, it is apparent that the Towers of

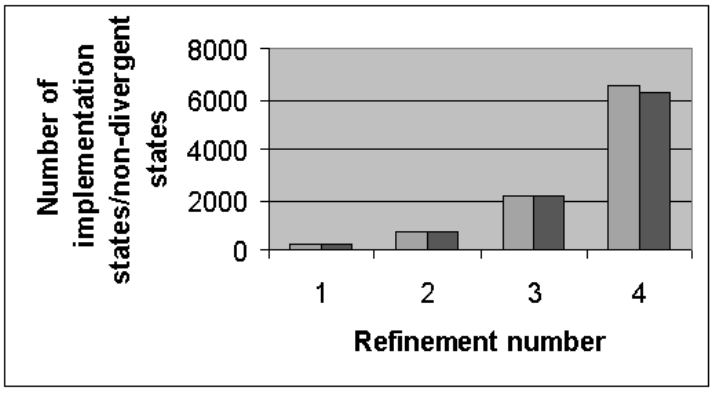


Figure 7.5: Fault-based analysis: System three

Hanoi is less amenable a system to fault seeding than the previous two systems. Again, the metrics are not appropriate for this kind of analysis.

7.6.4 System four: the dining philosophers

For System four, nine refinement runs were carried out, ranging from 1 to 9 philosophers. Table 7.11 contains the summary metrics for the nine refinement checks. Just as for the analysis described earlier in the chapter, the difficulty with System four was finding a suitable refinement check which would provide meaningful information about the specification and its implementation. What little information was available did not shed any light on the fault to be seeded in the process; in fact, we changed

```
PHIL = left.Up -> right.Down -> left.Up -> right.Down -> PHIL
```

to

```
PHIL = left.Up -> right.Down -> left.Down -> right.Down -> PHIL
```

The fault invested changed the second `left.Up` to `left.Down`. Table 7.11 reinforces the view that the original set of proposed metrics are inapplicable for fault-based analyses in both of

Metric	min.	max.	median	mean
Number of states in the specification	1	1	1	1
Number of states in the implementation	1	1	1	1
Number of non-divergent states in the implementation	1	1	1	1

Table 7.11: Summary metrics for fault-based analysis (System four)

the two Type II systems. We note that in Tables 7.8 to 7.11, the values of only three metrics have been given. This is because, based on inspection of the empirical data produced, these three showed most relevance for analysis in the context of faults.

7.7 Conclusions

It is clear from the empirical analysis carried out in this and the preceding chapter that there are significant differences between Type I and Type II CSP systems. The main conclusion to be drawn from the analysis contained in this chapter is that the metrics described in Chapter 5 are adequate for protocol-based systems only. For applications where there is significant problem-solving element, issues related to the applicability of metrics arise immediately. These issues range from limits on the type of refinement check possible, to a system-related problem of inadequate stack space to handle the level of recursion. It is clear from the analysis carried out that a separate set of metrics to characterise features of problem-solving CSP processes needs to be produced.

A number of alternative metrics were proposed which it was hoped would more accurately capture the features of problem-based CSP systems and these were collected using the example of a railway crossing application. In terms of its characteristics, this final application seems to lie somewhere between the two domains previously analysed, although further analyses need to be undertaken to confirm or refute this. The original proposed set

of metrics also proved inappropriate for any serious application of seeding faults into the implementation.

In the following chapter we take a critical look at the four CSP systems and the metrics thereof, justifying their use, suggesting ways in which the analysis could possibly have been improved, and how the work could be extended. We also look tentatively at the effect that slightly changing the specification can have on the metrics for the multiplexed buffer problem.

Chapter 8

Conclusions and future research

8.1 Introduction

In Chapter 1, we stated the problem that we were attempting to solve in this thesis; namely, the problem of ensuring that a piece of software matched the requirements expressed in a specification. In that chapter, we described how this became a problem of comparing a specification and its implementation, ensuring that implementation and specification were both expressed in such a way that this comparison could be made. Throughout the thesis, CSP and its underlying semantics have been used for all the systems analysed. The choice was deliberate, since it permitted us to use the same notation for both specification and implementation and hence achieve a transparent mapping between the two. The objectives of the thesis were:

1. To obtain a greater understanding of the behaviour of CSP systems as they are developed through the process of refinement. To this end, four CSP systems were analysed empirically. For each of the four corresponding specifications, an implementation or partial implementation of the specification was examined to determine the changing

behaviour of the implementation as the cardinality of the state space was increased; to effect such an increase, the range of data values used by the CSP systems was successively increased to a practically scalable size.

2. To assess the capability of a set of proposed metrics for measuring features of the specification and implementation during the process of refinement, with specific emphasis on the types of state encountered in both specification and implementation. To this end, a set of *refinement* metrics was proposed and evaluated on each of the four CSP systems. It was found that, while appropriate for Type I CSP systems, the initial set of metrics proved inappropriate for Type II CSP systems. A closer examination of the metrics was made and tentative new metrics more appropriate for the latter were proposed.

In the light of the content of the previous chapters and the experience this has given us, and taking the two objectives stated above into consideration, we next post-analyse the main contributions made in the thesis.

In the next section, the two main contributions are stated and justification for each of those contributions then given, together with suggestions as to what might have been changed had the analysis been repeated. In Section 8.3, we outline the main lessons learnt from the research contained herein and provide pointers to future research.

8.2 Contributions of the thesis

The two main contributions of the thesis are:

1. A greater understanding of the behaviour of implementation pertaining to the two types of application domain (Type I and Type II) has been acquired. It has been found that

this behaviour of CSP systems differs significantly across application domains.

2. A greater understanding of the requirements for metrics when applied to different application types, within a CSP context, has also been acquired. It has been found that metrics covering the process of refinement are domain-dependent. In other words, some metrics are applicable to Type I problems, while others are more applicable to Type II problems.

We next substantiate the above two statements.

8.2.1 Contribution one

The insights provided by our analysis of the four CSP systems studied can be looked at from two viewpoints – within the two application domains and between the two application domains.

From the first viewpoint, namely, within the two types of application domain, there were noticeable similarities between each pair of systems. When compared, the multiplexed buffers (System one) and the alternating-bit protocol (System two) showed only minor differences in terms of the upper limit on the number of states which could be generated and the nature of some of the other metrics collected (e.g., the number of sub-processes, which tended to remain static throughout refinement). Both of these systems were characterised by a strong emphasis on communication between sub-processes and an architecture amenable to decomposition through various levels of abstraction. Equally, for the final two systems, the Towers of Hanoi (System three) and The Dining Philosophers (System four), it was noticeable how similar these two systems were in terms of their features, namely, a dependance on the use of recursion to solve the problems they each addressed and a tendency to use the same CSP constructs.

From the second viewpoint, namely, between application domains, there was a stark difference between the first two systems and the second two systems. This included differences in the way the systems were written as well as other statically analysable differences (e.g., extent of hiding, level of decomposition). The two different application domains also tended to use different CSP syntactic constructs. Take, for example, the varying levels of relabelling over the four systems with none found in the first pair of systems and a high level in the second pair of systems. The number of states capable of being generated was also very small in the latter two systems, in contrast to the first two systems, and the opportunities for analysing refinement, via the metrics of Chapter 5, a good deal less. In the last two systems, it was more a case of checking the implementation for terminating conditions rather than against the specification.

8.2.2 Contribution two

Metrics were proposed which were thought to be applicable to the CSP environment, and hence applicable to all CSP systems. In substantiating our claim about this contribution, it is perhaps as valid to state the lessons that have been learnt in terms of the metrics, rather than simply outlining the development of the metrics themselves. With hindsight, one difficult, but interesting lesson learnt, was that the initial set of twelve metrics lacked flexibility; the majority of the metrics were found to be inapplicable for Type II problems. This raised a number of interesting questions.

Firstly, for a particular programming medium, is there a generally applicable subset of metrics? In the object-oriented community, this question has proved dificult to answer, and the indication from the research contained herein is that evaluating metrics for CSP systems is no less difficult.

Secondly, were the metric values obtained from each of the four CSP systems reflective of the nature of the problem, or simply the way that the CSP systems had been expressed?

Consider the last two CSP systems. Conceivably, these two systems could have been written in a style which minimised the level of recursion in the code, i.e., they could have been written in an iterative fashion. We need to ask whether the same metrics values would have been produced had those two systems been coded differently. In response to this, we emphasise that coding an implementation of the Towers of Hanoi or Dining Philosophers problem, without using recursive techniques, goes against the CSP ethos, which, after all, is based on the use of recursive techniques. It is thus likely that the metrics values obtained would not differ significantly whether the implementation was written recursively or iteratively, for the simple reason that certain CSP constructs (e.g., renaming and non-determinism) would have to be used in any implementation of these two problems. The only metrics values which might change are those related to the number of sub-processes generated, due to the level of recursion, by the specification and implementation of the problem. From a practical viewpoint, if the metrics values were different as a result, then this may merely indicate a poor design or poor choice of coding constructs in the implementation by the developer, rather than being indicative of a poor set of metrics.

Consider the first two CSP systems. It is difficult to envisage an alternative way of coding these systems, and so, again, the problem of having to decide whether alternatively coded implementations would produce different metrics values does not arise. Hence, the conclusion is that, to a large extent, it is the nature of the problem which determines the metrics values, and not the way that the CSP system is coded.

As to the question of what could have been done differently had the research been repeated, only minor aspects would be changed, and these mainly relate to future research

anyway.

Broadening the scope of the analysis to more CSP systems across other application domains would be an obvious extension to the research reported in this thesis.

Using principal components analysis to extract the most appropriate metrics across more than the two domains examined herein would be another avenue for future research, but would require more analysis of alternative metrics. It might also have been useful to automate the whole process of metrics collection and analysis. At present, only the collection is automated, which tends to slow down the extraction of results considerably. In terms of a wider context in which the set of proposed metrics can be placed, the current trend in software metrics is towards their use for predicting development effort and cost as early as possible in the development process [BW01, SC01, SK01]. The use of appropriate measurement and statistical techniques lies at the heart of this research. The emphasis of the set of metrics proposed herein was not to go this far; however, it is not inconceivable that they could be used for prediction purposes in a CSP system; this would be a topic for future research.

One area which was investigated briefly, as a final task of the analysis, was the effect of changing the specification of a CSP system on the metrics values. To this end, thirty-six refinement checks were carried out with a modified specification using System one as a basis. The original specification defined by

```
Copy(i) = left.i ? x -> right.i ! x -> Copy(i)

Spec = ||| i:Tag @ Copy(i)
```

was modified to include an extra `right ! x` in `Copy(i)` immediately following the initial `left.i ? x`. A number of scatter plots were then investigated. We describe only the key points to come out of this initial investigation.

Firstly, addition of this extra action rendered it impossible to achieve the limit of thirty-

seven refinement checks (as shown in Table 6.3), so only thirty-six refinement checks were carried out. This was due to the limitation of the machine's virtual memory. It was remarkable how addition of just one action to the specification caused such a change in the number of states generated by the specification (metric 1). The metrics values corresponding to row 1 of Table 6.1 now are: 3 (min.), 9261 (max.), 147 (median), 1265 (mean) and 2295.45 (std. dev.).

Secondly, related to the previous point, small changes to a CSP specification may give rise to the need for significant re-design of the refinement process. The low number of non-divergent states in the implementation (metric 10) observed for the new `Spec` when compared against the old `System` reflected this need. The metrics values corresponding to row 10 of Table 6.1 now are: 9 (min.), 30932 (max.), 585 (median), 3860 (mean) and 7201.52 (std. dev.).

Finally, some of the other metrics relating to the specification did not vary from their values in the original version of `Spec`. An example of this is the number of distinct actions in the specification (metric 3); another example is the number of sub-processes in the specification (metric 7). Being at a higher level of abstraction, it would seem that the specification was insensitive to certain types of change in its composition. A rigorous analysis of the nature of this set of changes would again be a topic for future research.

8.3 Lessons learnt and future research

A number of lessons, in terms of research techniques, have been learnt during the conduct of the research in this thesis.

Firstly, that it is as valid to report negative results as it is to report positive results. This relates, in particular, to the initial statement of the proposed metrics for Type I problems,

which were then pruned and expanded in order that they be applicable to Type II problems.

Secondly, that empirical work, carrying out analyses such as those in the thesis, reveals more questions to be answered than it actually answers. Parallels with other programming languages, the coding habits of developers, program comprehension and maintenance issues all have input to the field of empirical analysis. Useful results, however, are more likely to emerge from taking a small area and investigating it well rather than tackling a large area and covering that area poorly.

Thirdly, that it is important to state the assumptions on which the research rests unequivocally. In the case of this thesis, it was that the CSP medium and semantics were to be used throughout and that the specification captured the requirements exactly.

Finally, that an open mind to alternative theories should always be held in work of this nature. In the context of this research, further studies in the area need to be undertaken to shed light on current results and thinking.

One area of future research would therefore be to extend the empirical analysis to other CSP systems to confirm (or not) the findings reported herein. For example, we have not claimed that the proposed metrics were a definitive set. Further research could therefore focus on introducing other metrics applicable to other application domains. As stated earlier (Section 8.2.2), one of the major lessons learnt throughout this thesis has been that empirical analysis requires a great deal of flexibility in terms of examining, refining and evolving metrics from first-cut versions to those which truly reflect the features of CSP systems. Future research would also address fault injection issues (and the effect this would have on the metrics values) in more depth than has so far been considered. For example, it might be fruitful to examine the effect of different fault *categories* in a CSP context, so building up a picture of the effect that a particular type of error by developers may have on an

implementation.

In view of the fact that we also looked briefly at the effect on the metrics values induced by changing the specification, future research will, again, need to delve in depth at issues related to changing requirements. Again, it may be fruitful to examine the effect of different categories of requirement on a CSP specification (and hence the CSP implementation). Finally, it may be useful to be able to predict attributes of CSP systems (cf. [SC01, SK01]) such as the number of generated states or number of sub-processes, based on the static features of those CSP systems (e.g., the data types). This would obviate the need for time-consuming refinement checks to be undertaken.

Appendix A

System one (Multiplexed Buffers)

```
-- Multiplexed buffers, version for fdr.1.1   -- Bill Roscoe
-- Modified for fdr.1.2 10/8/92 Dave Jackson

-- The idea of this example is to multiplex a number of buffers down a
-- pair of channels.  They can all be in one direction, or there might be
-- some both ways.  The techniques demonstrated here work for all
-- numbers of buffers, and any types for transmission.  The number of states
-- in the system can be easily increased to any desired size by increasing
-- either the number of buffers, or the size of the transmitted type.

datatype Tag  = t1 | t2 | t3
datatype Data = d1 | d2

channel left, right : Tag.Data
channel snd_mess, rcv_mess : Tag.Data
channel snd_ack, rcv_ack   : Tag
channel mess : Tag.Data
channel ack  : Tag

-- The following four processes form the core of the system
--
--
--    --> SndMess -->  ...........   --> RcvMess -->
--
--    <-- RcvAck  <--  ...........   <-- SndAck  <--
--
-- SndMess and RcvMess send and receive tagged messages, while
-- SndAck and RcvAck send and receive acknowledgements.

SndMess = [] i:Tag @ (snd_mess.i ? x -> mess ! i.x -> SndMess)
```

```
RcvMess = mess ? i.x -> rcv_mess.i ! x -> RcvMess

SndAck = [] i:Tag @ (snd_ack.i  -> ack ! i -> SndAck)

RcvAck = ack ? i -> rcv_ack.i -> RcvAck

-- These four processes communicate with equal numbers of transmitters (Tx)
-- and receivers (Rx), which in turn provide the interface with the
-- environment.

Tx(i) = left.i ? x -> snd_mess.i ! x -> rcv_ack.i -> Tx(i)

Rx(i) = rcv_mess.i ? x -> right.i ! x -> snd_ack.i -> Rx(i)

FaultyRx(i) = rcv_mess.i ? x -> right.i ! x ->(FaultyRx(i)
                                              |~| snd_ack.i -> FaultyRx(i))

-- Txs is the collection of transmitters working independently.

Txs = ||| i:Tag @ Tx(i)

-- LHS is just everything concerned with transmission combined, with
-- internal communication hidden.

LHS = (Txs [|{|snd_mess, rcv_ack|}|](SndMess ||| RcvAck))\{|snd_mess, rcv_ack|}

-- The receiving side is built in a similar way.

Rxs = ||| i:Tag @ Rx(i)

FaultyRxs = Rx(t1) ||| Rx(t2) ||| FaultyRx(t3)

RHS = (Rxs [|{|rcv_mess, snd_ack|}|]
                  (RcvMess ||| SndAck))\{|rcv_mess, snd_ack|}

FaultyRHS = (FaultyRxs [|{|rcv_mess, snd_ack|}|]
                        (RcvMess ||| SndAck))\{|rcv_mess, snd_ack|}

-- Finally we put it all together, and hide internal communication.

System = (LHS [|{|mess, ack|}|] RHS)\{|mess,ack|}

FaultySystem = (LHS [|{|mess, ack|}|] FaultyRHS)\{|mess, ack|}

-- The specification is just the parallel composition of several one-place
-- buffers.

Copy(i) = left.i ? x -> right.i ! x -> Copy(i)

Spec = ||| i:Tag @ Copy(i)
```

```
-- Correctness of the system is asserted by  Spec [FD= System.

assert Spec [FD= System

-- If the multiplexer is being used as part of a larger system, then
-- it would make a lot of sense to prove that it meets its specification
-- and then use its specification in its stead in higher-level system
-- descriptions.  This applies even if the higher-level system does not
-- break up into smaller components, since the state-space of the
-- specification is significantly smaller than that of the multiplexer,
-- which will make the verification of a large system quicker.  It is
-- even more true if the channels of the multiplexer are used independently,
-- in other words if each external channel of the multiplexer is connected
-- to a different user, and the users do not interact otherwise,
-- for it would then be sufficient to prove that each of the separate
-- pairs of processes interacting via our multiplexer is correct relative
-- to its own specification, with a simple one-place buffer between them.

-- For we would have proved the equivalence, by the correctness of the
-- multiplexer, of our system with a set of three-process independent ones.
```

Appendix B

System two (Alternating Bit Protocol)

```
{-
  Alternating bit protocol.

  Bill Roscoe, August 1992
  Adapted for FDR2.11, Bryan Scattergood, April 1997

  This is the initial example of a set which make use of a pair of media
  which are permitted to lose data, and provided no infinite sequence is lost
  will work independently of how lossy the channels are (unlike the file
  prots.csp where the protocols were designed to cope with specific badnesses)
  They work by transmitting messages one way and acknowledgements the other.

  The alternating bit protocol provides the most standard of all protocol
  examples.  The treatment here has a lot in common with that in the
  Formal Systems information leaflet "The Untimed Analysis of Concurrent
  Systems".
-}

{-
  Channels and data types
  left and right are the external input and output.
  a and b carry a tag and a data value.
  c and d carry an acknowledgement tag.
  (In this protocol tags are bits.)

              a  PUT  b
```

```
  left        /     \       right
  ------> SEND       RECV ------>
              \     /
            d  GET  c
-}

DATA = {2,3}  -- in a data-independent program, where nothing is done to
              -- data, or is conditional on data, this is sufficient to
              -- establish correctness.

channel left,right : DATA
channel a, b : Bool.DATA
channel c, d : Bool

{-
  The overall specification we want to meet is that of a buffer.
-}

SPEC = let
    {-
      The most nondeterministic (left-to-right) buffer with size bounded
      by N is given by BUFF(<>, N), where
    -}
    BUFF(s, N) =
      if null(s) then
        left?x -> BUFF(<x>, N)
      else
        right!head(s) -> BUFF(tail(s), N)
        []
        #s < N & (STOP |~| left?x -> BUFF(s^<x>, N))
    {-
      For our purposes we will set N = 3 since this example does not introduce
      more buffering than this.
    -}
  within BUFF(<>, 3)

{-
  The protocol is designed to work in the presence of lossy channels.
  We specify here channels which must transmit one out of any three values, but
  any definition would work provided it maintains order and does not lose
  an infinite sequence of values.  The only difference would evidence itself
  in the size of the state-space!
-}

lossy_buffer(in, out, bound) =
  let
    -- Increasing bound makes this definition less deterministic.
    -- n is the number of outputs which may be discarded.
    B(0) = in?x -> out!x -> B(bound-1)
    B(n) = in?x -> (B(n-1) |~| out!x -> B(bound-1))
  within B(bound-1)
```

```
PUT = lossy_buffer(a, b, 3)

GET = lossy_buffer(c, d, 3)

{-
  The implementation of the protocol consists of a sender process and
  receiver process, linked by PUT and GET above.
-}

SEND =
  let
    Null = 99 -- any value not in DATA
    {-
      The sender process is parameterised by the current value it
      tries to send out, which may be Null in which case it does
      not try to send it, but instead accepts a new one from
      channel left.
      It is always willing to accept any acknowledgement, and if
      the tag corresponds to the current bit, v is made Null.
    -}
    S(v,bit) =
      (if v == Null then left?x -> S(x, not bit) else a!bit!v -> S(v, bit))
      []
      d?ack -> S(if ack==bit then Null else v, bit)
    {-
      Initially the data value is Null and bit is true so the first
      value input gets bit false.
    -}
  within S(Null, true)

RECV =
  let
    {-
      The basic part of the receiver takes in messages, sends
      acknowledgements, and transmits messages to the environment.
      R(b) is a process that will always accept a message or
      send an acknowledgement, save that it will not do so when it
      has a pending message to transmit to the environment.
    -}
    R(bit) =
      b?tag?data -> (if tag==bit then right!data -> R(not bit) else R(bit))
      []
      c!not bit -> R(bit)
    {-
      The first message to be output has tag false, and there is no pending
      message.
    -}
  within R(false)

{-
```

```
  If this receiver is placed in the system, there is the danger of
  livelock, or divergence, if an infinite sequence of acknowledgements is
  transmitted by RECV and received by SEND without the next message being
  transmitted, as is possible.  Alternatively, a message can be transmitted
  and received infinitely without being acknowledged.
  Thus, while the following system is
  partially correct, it can diverge (infinitely):
-}

make_system(receiver) =
  make_full_system(SEND, PUT|||GET, receiver)

make_full_system(sender, wiring, receiver) =
  sender[|{|a,d|}|](wiring[|{|b,c|}|]receiver)\{|a,b,c,d|}

DIVSYSTEM = make_system(RECV)

assert DIVSYSTEM :[livelock free]

{-
  We can avoid divergence by preventing the receiver acknowledging or
  receiving infinitely without doing the other (acknowledging). This can
  be done by putting it in parallel with any process which allows these
  actions in such a way as to avoid these infinitely unfair sequences.
  In fact, the receiver may choose one of the two to do rather than give
  the choice as above.  Examples that will work are:
-}

-- Simple alternation

ALT = b?_ -> c?_ -> ALT

-- Give the environment the choice, provided there is no run of more
-- than M of either.

LIMIT(M) =
  let
    L(bs,cs) =
      bs < M & b?_ -> L(bs+1, 0)
      []
      cs < M & c?_ -> L(0, cs+1)
  within L(0,0)

-- Choose nondeterministically which to allow, provided the totals
-- of b's and c's do not differ too much.

NDC(M) =
  let
    C(n) =
      if n==0 then
        c?_ -> C(n+1)
```

```
      else if n==M then
        b?_ -> C(n-1)
      else
        c?_ -> C(n+1) |~| b?_ -> C(n-1)
  within C(M/2)

-- Modified receiver processes, with small values for the constants, are

modify_receiver(constraint) =
  RECV [|{|b,c|}|] constraint

RCVA = modify_receiver(ALT)
RCVL = modify_receiver(LIMIT(3))
RCVN = modify_receiver(NDC(4))

-- and the checks of the respective systems against SPEC

assert SPEC [FD= make_system(RCVA)
assert SPEC [FD= make_system(RCVL)
assert SPEC [FD= make_system(RCVN)

{-
  Of course, one would not normally construct one's receiver as a composition
  of an algorithmic process and constraint in this way, but we now know that
  any receiver which refines RCVN will work.  For example,
-}

RCVimp =
  let
    R(bit) =
      b?tag?data -> if tag==bit then
                      right!data -> c!tag -> R(not bit)
                    else
                      c!tag -> R(bit)
  within R(false)

{-
  You can check that RCVimp refines RCVN, which proves that the larger
  check below is correct.  (This can, in this instance, be proved directly.)
-}

assert RCVN [FD= RCVimp

assert SPEC [FD= make_system(RCVimp)

{-
  Indeed, RCVimp actually equals (semantically) RCVA, and you can check
  that refinement either way.
-}

assert RCVimp [FD= RCVA
```

```
assert RCVA [FD= RCVimp

{-
  If you want to develop this example much further, perhaps by inserting
  a more interesting process in one or both channels, the state-space may
  grow uncomfortably large for a full check (including absence of divergence).

  Any different channel definitions which satisfy all of

  1. outputs(tr)  subseq of  inputs(tr)  in the obvious sense

  2. will not do an infinite sequence of inputs without an output

  3. refines LIVCH (given below) and hence can always either input
     any value, or make an output

  are substitutable for PUT and/or GET.
-}

LIVCH(in, out) =
  let
    P =
      in?_ -> P
      |~|
      |~| x:{|out|} @ x -> P
  within P

assert LIVCH(a,b) [FD= PUT
assert LIVCH(c,d) [FD= GET

{-
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   Failures-only Checking
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  The following is fairly advanced material for those who want to
  understand the subtle differences between full and failures-only, and even
  exploit them.

  As we know, Failures and Traces checking do not test the implementation
  for potential divergence.  This is usually something to overcome
  by establishing separately that the implementation is divergence-free.

  Provided we know clearly what we are doing, there are cases where
  establishing facts about processes which can diverge are valuable.
  In other words, there are useful theorems that can be proved using
  these restricted forms of check that are not provable using a full check.

  We know of two sorts of results to prove in this way.  They are
  similar in execution, but rather different in interpretation, and
  both can be successfully demonstrated using the alternating bit
```

```
  example.

  Spec [=F Imp  proves that every trace, and every stable refusal
  of Imp is satisfactory.  If we take a divergence-free
  component of Imp and refine it, then not only will the
  overall set of failures and divergence be refined (reduced)
  but so will the observable traces and stable refusals.

  It follows that, if we can establish Spec  [=F Imp, then we have proved
  a theorem that any Imp' produced by refining the component which is
  also divergence-free (i.e., Imp' is), then Imp' will failures-divergence
  refine Spec.

  This is valuable in arguments like that involving the receiver process
  above, where a number of different processes were introduced, as varying
  ways of eliminating divergence.  We proved each of them worked,
  but really we would like a general result about a class of receiver
  processes, which leaves divergence as the only remaining issue.  This
  can be done with the generalised co-process
-}

ND = b?_ -> ND |~| c?_ -> ND

{-
  which is an approximation to  ALT, LIMIT and NDC above. The generalised
  receiver process
-}

RCVG = modify_receiver(ND)

{-
  approximates any conceivable process we might want to put at the
  right-hand-side of the alternating bit protocol.  Note that it can
  always choose whether it wants to accept a message or send an acknowledgement
-}

-- The system

GSYSTEM = make_system(RCVG)

{-
  will be found to diverge if you use a full check.  However the failures
  check will work, which, as we said above, establishes that
  any refinement of RCVG which eliminates divergence from the whole
  system will give a valid failures-divergence refinement of SPEC.

  Note that we could not have made the same argument using RECV, since
  it is a deterministic process.  In fact, DIVSYSTEM has no stable
  states at all, a very different proposition from GSYSTEM,
  which implicitly has all the stable states of every possible working
  right-hand process.
```

```
-}

assert SPEC [F= GSYSTEM
assert SPEC [FD= GSYSTEM

{-
  The second sort of result uses very similar techniques to reason about
  "fairness" issues.  It perhaps does not make much sense to talk about a
  fair receiver process, when the design of that process is likely to be
  firmly within our control.  But it certainly makes sense to introduce
  fairness assumptions about the communications medium.  It is really more
  natural to bring in assumption like "the medium will never lose an infinite
  sequence of consecutive messages" rather than "it will not lose more than
  three out of every four".  The ideal model of a communications medium
  in this respect is
-}

PUT' = a?tag?data -> (b!tag!data -> PUT' |~|  PUT')

GET' = c?tag -> (d!tag -> GET' |~|  GET')

{-
  with the additional assumption that an infinite set of messages is not
  lost.  This actually defines perfectly valid processes in the infinite
  traces/failures model of CSP, and mathematical arguments in that model
  will exclude divergence from a system such as
-}

FSYSTEM =
  make_full_system(SEND, PUT'|||GET', RCVA)

{-
  with the additional assumptions about EG and FG that we cannot make in
  the finite failures/divergence model.  Since the additional assumptions
  only serve to refine EG and FG further, proving that FSYSTEM refines
  SPEC using a failures check will prove that the system with the
  fairness assumptions in place will be a full refinement of SPEC.
-}

assert SPEC [F= FSYSTEM
```

Appendix C

System three (Towers of Hanoi)

```
{-
  A version of the Towers of Hanoi using lots of features
  which were not present in FDR 1.4
  JBS 6 March 1995 (based loosely on AWR's version for FDR 1.4)
-}

transparent diamond

n             = 5 -- How many discs

-- Discs are numbered
DISCS         = {1..n}

-- But the pegs are labelled
datatype PEGS = A | B | C

{-
  For a given peg, we can get a new disc or put the
  top disc somewhere else.  We are also allowed to
  to indicate when the peg is full.
-}

channel get, put : DISCS
channel full

-- We are allowed to put any *smaller* disc onto the current stack
allowed(s) = { 1..head(s^<n+1>)-1 }

PEG(s) =
  get?d:allowed(s)->PEG(<d>^s)
```

```
    []
  not null(s) & put!head(s)->PEG(tail(s))
    []
  length(s) == n & full->PEG(s)

{-
  Now, given a simple peg we can rename it to form each
  of the three physical pegs (‘poles’) of the puzzle.

  move.d.i.j indicates that disc d moves to pole i from pole j
-}

channel move : DISCS.PEGS.PEGS
channel complete : PEGS

initial(p)   = < 1..n | p == A >

POLE(p) =
  PEG(initial(p))
    [[ full <- complete.p,
       get.d <- move.d.p.i,
       put.d <- move.d.i.p | i<- PEGS, i != p, d<-DISCS ]]

{-
  The puzzle is just the three poles, communicating on the
  relevant events: all the moves, and the done/notdone events.
-}

interface(p) = { move.d.p.i, complete.p | d<-DISCS, i<-PEGS }

PUZZLE =
  || p : PEGS @ [ interface(p) ] diamond(POLE(p))

{-
  The puzzle is solved by asserting that C cannot become complete.
  Then the trace that refutes the assertion is the solution.
-}

NOTSOLVED = complete?x:{A,B}-> NOTSOLVED [] move?x -> NOTSOLVED

assert NOTSOLVED [T= PUZZLE
assert PUZZLE \ {| complete.A, complete.B, move |} [F= STOP
```

Appendix D

System four (Dining Philosophers)

```
-- Play with massive numbers of dining philosophers
-- (Powers of 2 only)
-- JBS, May 1997.

datatype Move = Up | Down

channel left, right : Move

FORK = left.Down->left.Down->FORK [] right.Down->right.Down->FORK

PHIL = left.Up->right.Down->left.Up->right.Down->PHIL

LPHILS(n) =
  let
    L(0) =
      FORK [right<->left] PHIL
    L(n) =
      let
        HALF = LPHILS(n-1)
      within HALF [right<->left] HALF
    transparent normal
  within normal(L(n) [[ ]])

RPHILS(n) =
  LPHILS(n) [[ left <- right, right <- left ]]

PHILS(n) =
  LPHILS(n-1) [|{| left, right |}|] RPHILS(n-1)

-- PHILS(n) represents a network of 2^n philosophers
```

```
X = PHILS(4)

assert X [FD= X
```

Appendix E

Railway Crossing

```
-- Model of a level crossing gate for FDR: revised version
-- Illustrating discrete-time modelling using untimed CSP

-- (c) Bill Roscoe, November 1992 and July 1995
-- Revised for FDR 2.11 May 1997

{-
  This file contains a revised version, to coincide with my 1995
  notes, of the level crossing gate example which was the first CSP
  program to use the "tock" model of time.

  The present version has (I think) a marginally better incorporation
  of timing information.
-}

-- The tock event represents the passing of a unit of time

channel tock

-- The following are the communications between the controller process and
-- the gate process

datatype GateControl = go_down | go_up | up | down

-- where we can think of the first two as being commands to it, and the
-- last two as being confirmations from a sensor that they are up or down.

channel gate : GateControl

-- For reasons discussed below, we introduce a special error event:
```

```
channel error

-- To model the speed of trains, and also the separation of more than one
-- train, we divide the track into segments that the trains can enter or
-- leave.

Segments = 10 -- the number of segments including the outside one
LastSeg = Segments - 1
TRACKS = {0..LastSeg}
REALTRACKS = {1..LastSeg}

-- Here, segment 0 represents the outside world, and [1,Segment) actual
-- track segments; including the crossing, which is at

GateSeg=3

-- This model handles two trains

datatype TRAINS = Thomas | Gordon

-- which can move between track segments

channel enter, leave : TRACKS.TRAINS

-- Trains are detected when they enter the first track segment by a sensor,
-- which drives the controller, and are also detected by a second sensor
-- when they leave GateSeg

datatype sensed = in | out

channel sensor : sensed


-- The following gives an untimed description of Train A on track segment j
-- A train not currently in the domain of interest is given index 0.

Train(A,j) =  enter.((j+1)%Segments).A -> leave.j.A -> Train(A,(j+1)%Segments)

-- There is no direct interference between the trains

Trains = Train(Thomas,0) ||| Train(Gordon,0)

-- The real track segments can be occupied by one train at a time, and each
-- time a train enters segment 1 or leaves GateSeg the sensors fire.

Track(j) =
  let
    Empty   = enter.j?A -> if j==1 then sensor.in -> Full(A) else Full(A)
    Full(A) = leave.j.A -> if j==GateSeg then sensor.out -> Empty else Empty
  within Empty
```

```
-- Like the trains, the untimed track segments do not communicate with
-- each other

Tracks = ||| j : REALTRACKS @ Track(j)

-- And we can put together the untimed network, noting that since there is
-- no process modelling the outside world there is no need to synchronise
-- on the enter and leave events for this area.

Network = Trains [|{|enter.j, leave.j | j<-REALTRACKS|}|] Tracks

-- We make assumptions about the speed of trains by placing (uniform)
-- upper and lower "speed limits" on the track segments:

-- MinTocksPerSeg = 3 -- make this a parameter to experiment with it
SlowTrain = 4       -- inverse speed parameter, MinTocksPerSegment
NormalTrain = 3
FastTrain = 2

MaxTocksPerSeg = 6

-- The speed regulators express bounds on the times between successive
-- enter events.

SpeedReg(j,MinTocksPerSeg) =
  let
    Empty   = enter.j?A -> Full(0) [] tock -> Empty
    Full(n) = n <  MaxTocksPerSeg & tock -> Full(n+1)
           [] MinTocksPerSeg <= n & enter.(j+1)%Segments?A -> Empty
  within Empty

-- The following pair of processes express the timing contraint that
-- the two sensor events occur within one time unit of a train entering
-- or leaving the domain.

InSensorTiming = tock -> InSensorTiming
              [] enter.1?A -> sensor.in -> InSensorTiming

OutSensorTiming = tock -> OutSensorTiming
               [] leave.GateSeg?A -> sensor.out -> OutSensorTiming

-- The timing constraints of the trains and sensors are combined into the
-- network as follows, noting that no speed limits are used outside the domain:

SpeedRegs(min) =
  || j : REALTRACKS @ [{|tock, enter.j, enter.(j+1)%Segments|}] SpeedReg(j,min)

SensorTiming = InSensorTiming [|{tock}|] OutSensorTiming

NetworkTiming(min) = SpeedRegs(min) [|{|tock, enter.1|}|] SensorTiming
```

```
TimedNetwork(min) =
  Network [|{|enter, sensor, leave.GateSeg|}|] NetworkTiming(min)

-- The last component of our system is a controller for the gate, whose duties
-- are to ensure that the gate is always down when there is a train on the
-- gate, and that it is up whenever prudent.

-- Unlike the first version of this example, here we will separate the
-- timing assumptions about how the gate behaves into a separate process.
-- But some timing details (relating to the intervals between sensors
-- firing and signals being sent to the gate) are coded directly into this
-- process to illustrate a different coding style to that used above:

Controller =
  let
    -- When the gate is up, the controller does nothing until the sensor
    -- detects an approaching train.
    -- In this state, time is allowed to pass arbitrarily, except that the
    -- signal for the gate to go down is sent immediately on the occurrence of
    -- the sensor event.
    ControllerUp = sensor.in -> gate!go_down -> ControllerGoingDown(1)
                [] sensor.out -> ERROR
                [] tock -> ControllerUp
    -- The two states ControllerGoingDown and ControllerDown (see below)
    -- both keep a record of how many trains have to pass before the gate
    -- may go up.
    -- Each time the sensor event occurs this count is increased.
    -- The count should not get greater than the number of trains that
    -- can legally be between the sensor and the gate (which equals
    -- the number of track segments).
    -- The ControllerGoingDown state comes to an end when the
    -- gate.down event occurs
    ControllerGoingDown(n) =
         (if GateSeg < n then ERROR else sensor.in -> ControllerGoingDown(n+1))
      [] gate.down -> ControllerDown(n)
      [] tock -> ControllerGoingDown(n)
      [] sensor.out -> ERROR
    -- When the gate is down, the occurrence of a train entering its
    -- sector causes no alarm, and each time a train leaves the gate
    -- sector the remaining count goes down, or the gate is signalled
    -- to go up, as appropriate.
    -- Time is allowed to pass arbitrarily in this state, except that
    -- the direction to the gate to go up is instantaneous when due.
    ControllerDown(n) =
         (if GateSeg < n then ERROR else sensor.in -> ControllerDown(n+1))
      [] sensor.out -> (if n==1 then gate!go_up -> ControllerGoingUp
                                else ControllerDown(n-1))
      [] tock -> ControllerDown(n)
    -- When the gate is going up, the inward sensor may still fire,
    -- which means that the gate must be signalled to go down again.
    -- Otherwise the gate goes up after UpTime units.
```

```
    ControllerGoingUp =  gate!up -> ControllerUp
                      [] tock -> ControllerGoingUp
                      [] sensor.in -> gate!go_down -> ControllerGoingDown(1)
                      [] sensor.out -> ERROR
  within ControllerUp

-- Any process will be allowed to generate an error event, and since we will
-- be establishing that these do not occur, we can make the successor process
-- anything we please, in this case STOP.

ERROR = error -> STOP

-- The following are the times we assume here for the gate to go up
-- and go down.  They represent upper bounds in each case.

-- DownTime = 5 -- make this a parameter for experimentation
VeryFastGate = 3
FastGate = 4
NormalGate = 5
SlowGate = 6

UpTime = 2

Gate(DownTime) =
  let
    GateUp = gate.go_up -> GateUp
          [] gate.go_down -> GateGoingDown(0)
  [] tock -> GateUp
    GateGoingDown(n) =
         gate.go_down -> GateGoingDown(n)
      [] if n == DownTime
         then gate.down -> GateDown
 else gate.down -> GateDown |~| tock -> GateGoingDown(n+1)
    GateDown = gate.go_down -> GateDown
            [] gate.go_up -> GateGoingUp(0)
    [] tock -> GateDown
    GateGoingUp(n) = gate.go_up -> GateGoingUp(n)
                   [] gate.go_down -> GateGoingDown(0)
                   [] if n == UpTime
                      then gate.up -> GateUp
              else gate.up -> GateUp |~| tock -> GateGoingUp(n+1)
  within GateUp

-- Since Gate has explicitly nondeterministic behaviour, we can expect
-- to gain by applying a compression function, such as diamond, to it;
-- we declare a number of "transparent" compression functions

transparent sbisim
transparent normalise
transparent explicate
transparent diamond
```

```
GateAndController(dt) = Controller [|{|tock,gate|}|] diamond(Gate(dt))

-- Finally, we put the network together with the gate unit to give our
-- overall system

System(invmaxspeed,gatedowntime) =
  TimedNetwork(invmaxspeed) [|{|sensor,tock|}|] GateAndController(gatedowntime)


-- And now for specifications.  Since we have not synchronised on any
-- error events, they would remain visible if they occurred.  Their
-- absence can be checked with

NoError = CHAOS(diff(Events,{error}))


-- This shows that none of the explicitly caught error conditions arises,
-- but does not show that the system has the required safety property of
-- having no train on the GateSeg when the gate is other than down.

-- The required specifications are slight generalisations of those
-- discussed in specs.csp; the following notation and development is
-- consistent with that discussed therein.

SETBETWEENx(EN,DIS,C) = ([]x:EN @ x -> SETOUTSIDEx(DIS,EN,C))
                        [] ([] x:DIS @ x -> SETBETWEENx(EN,DIS,C))

SETOUTSIDEx(DIS,EN,C) = ([] c:C @ c -> SETOUTSIDEx(DIS,EN,C))
                        [] ([] x: EN @ x -> SETOUTSIDEx(DIS,EN,C))
                        [] ([] x:DIS @ x -> SETBETWEENx(EN,DIS,C))

-- The above capture the sort of relationships we need between the
-- relevant events.  If we want to stay within Failures-Divergence Refinement
-- (as opposed to using Trace checking subtly), we need to do the following to
-- turn them into the conditions we need:

EnterWhenDown =
  SETBETWEENx({gate.down},
              {gate.up,gate.go_up,gate.go_down},
              {|enter.GateSeg|})
  [|{|gate, enter.GateSeg|}|]
  CHAOS(Events)

GateStillWhenTrain =
  SETOUTSIDEx({|enter.GateSeg|},{|leave.GateSeg|},{|gate|})
  [|{|gate,enter.GateSeg,leave.GateSeg|}|]
  CHAOS(Events)

-- So we can form a single safety spec by conjoining these:
```

```
Safety = EnterWhenDown [|Events|] GateStillWhenTrain

-- There are a number of possible combinations which may be of interest.

-- An important form of "liveness" we have thus far ignored is that the clock
-- is not stopped: for this it is sufficient that TimingConsistency
-- refines TOCKS, where

TOCKS = tock -> TOCKS

-- The following is the set of events that we can rely on the environment
-- for delaying them.

Delayable = {|enter.1|}
NonTock = diff(Events,{tock})
TimingConsistency(ts,gs) =
  explicate(System(ts,gs)[|Delayable|]normalise(CHAOS(Delayable))\NonTock)

assert TOCKS [FD= TimingConsistency(NormalTrain,NormalGate)

-- The safety condition completely ignored time (although, if you change some
-- of the timing constants enough, you will find it relies upon timing for
-- it to be satisfied).  Because of the way we are modelling time, the
-- main liveness constraint (that the gate is up when prudent) actually
-- becomes a safety condition (one on traces).  It is the combination of this
-- with the TOCKS condition above (asserting that time passes) that gives
-- it the desired meaning.

-- We will specify that when X units of time have passed since the last
-- train left the gate, it must be open, and remain so until another
-- train enters the system.  This is done by the following,  which monitor
-- the number of trains in the system and, once the last train has left, no
-- more than X units of time pass (tock events) before the gate is up.  The
-- gate is not permitted to go down until a train is in the system.

Liveness(X) =
  let
    Idle = tock -> Idle
        [] enter.1?_ -> Busy(1)
    Busy(n) = tock -> Busy(n)
           [] enter.1?_ -> Busy(if n < GateSeg then (n+1) else n)
           [] leave.GateSeg?_ -> (if n==1 then UpBefore(X) else Busy(n-1))
           [] gate?_ -> Busy(n)
    UpBefore(m) = m != 0 & tock -> UpBefore(m-1)
               [] gate?x -> (if x==up then Idle else UpBefore(m))
               [] enter.1?_ -> Busy(1)
  -- Initially the gate is up in the system, so the liveness condition
  -- takes this into account.
  within Idle

GateLive(X) = Liveness(X) [|{|tock,gate,enter.1,leave.GateSeg|}|]CHAOS(Events)
```

Appendix F

Calculation of R-squared value

In fitting equations to data, a measure of the *goodness-of-fit*, designated by *Rsv*, is given by

$$R^2 = 1 - (SSE/SST)$$

where

$$SSE = \sum (Y_i - \hat{Y}_i)^2 \text{ and } SST = \sum Y_i^2 - (\sum Y_i)^2 \ / \ n$$

References

[Abr96] J.-R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AD94] G.D. Abowd and A.J. Dix. Integrating status and event phenomena in formal specifications of interactive processes. In *SIGSOFT'94, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, USA*, pages 44–52, 1994.

[AG94a] R. Allen and D. Garlan. Formal connectors. Technical report, CMU-CS-94-115, March 1994.

[AG94b] R. Allen and D. Garlan. Formalising architectural connection. In *ICSE'94, Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy*, pages 71–80, 1994.

[AH92] L. Aceto and M. Hennessy. Termination, deadlock and divergence. *Journal of the ACM*, 39(1):147–187, 1992.

[AH93] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. *Information and Computation*, 103:204–269, 1993.

[AH00] S. Antoy and D. Hamlet. Automatically checking an implementation against its

formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, 2000.

[AM94] B. Arrowsmith and B. McMillin. CCSP - a formal system for distributed program debugging. Technical report, University of Missouri - Rolla, June 1994.

[Arn94] A. Arnold. *Finite Transition Systems.* Prentice Hall, Hemel Hempstead, 1994.

[Bac86] R.C. Backhouse. *Program Construction and Verification.* Prentice Hall, Hemel Hempstead, 1986.

[Bar93] J.G.P. Barnes. *Programming in Ada, Fourth Edition.* Addison-Wesley, Reading, Massachusetts, 1993.

[BBM96] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[BDFM99] R. BenAyed, J. Desharnais, M. Frappier, and A. Mili. Mathematical foundations for program transformations. In *Lecture Notes in Computer Science, Volume 1559, editor P. Flener*, pages 319–321, 1999.

[BDM97] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97), Boston, USA*, pages 412–421, 1997.

[BFVY96] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31:560–599, 1984.

[BJ82] D. Bjorner and C.B. Jones. *Formal Specification and Software Development.* Prentice Hall International, London, 1982.

[Boe81] B.W. Boehm. *Software Engineering Economics.* Prentice Hall, New York, 1981.

[BR88] V.R. Basili and H.D. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, 1988.

[Bra90] I. Bratko. *Prolog, Programming for Artificial Intelligence.* Addison Wesley, Wokingham, England, 1990.

[Bro83] S. D. Brookes. *A Model for Communicating Sequential Processes.* Ph.D. dissertation, Oxford University, University College, January 1983.

[BS94] B. Berthomieu and T. Le Sergent. Programming behaviors in an ML framework - the syntax and semantics of LCS. In *Lecture Notes in Computer Science, Volume 788, editor D. Sannella*, pages 89–104, 1994.

[BW01] L. Briand and J. Wust. Modeling development effort in object-oriented systems using design properties. *IEEE Transactions on Software Engineering*, 27(11):963–986, 2001.

[BWW91] J. Bainbrdge, R. Whitty, and J. Wordsworth. Obtaining structural metrics of Z specifications for systems development. In *Z User Workshop '90, editor J.E. Nicholls, Oxford*, pages 269–281, 1991.

[CB84] D.J. Cooke and H.E. Bez. *Computer Mathematics.* Cambridge University Press, Cambridge, 1984.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):467–493, 1994.

[CM96] S.J. Counsell and K.L. Mannock. Comparison of a specification and its implementation using fdr. In *Proceedings 1996 International Workshop on Dependability in Advanced Computing Paradigms, Hitachi, Japan, IEEE Computer Society*, pages 663–680, 1996.

[CNM00] S Counsell, P Newson, and E Mendes. Architectural level hypothesis testing through reverse engineering of object-oriented software. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC'2000), Limerick, Ireland*, pages 60–66, 2000.

[Cou92] S.J. Counsell. Software reliability issues in CCS and CSP. In *Proceedings First International Conference Software Quality Management, Southampton, UK*, pages 663–680, 1992.

[CS00] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. 26(8):786–796, 2000.

[CW98] J. E. Cook and A. L. Wolf. Balboa: A framework for event-based process data analysis. Technical report, University of Colorado, CU-CS-851-98, 1998.

[Dep81] United States Defense Department. *Programming Language Ada: Reference Manual, Lecture Notes in Computer Science, Volume 106.* New York, 1981.

[Dil95] A. Diller. *Z: An Introduction to Formal Methods, Second Edition.* John Wiley and Sons, Chichester, 1995.

[DY94] L.K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *SIGSOFT'94, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, USA*, pages 140–153, 1994.

[Fen94] N.E. Fenton. Software measurement: a necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, 1994.

[For97] Formal Systems (Europe) Ltd. *Failures Divergence Refinement — User Manual and Tutorial*, version 2.28 edition, October 1997.

[FP96] N. Fenton and S.L. Pfleeger. *Software Metrics, A Rigorous and Practical Approach.* International Thomson Computer Press, London, 1996.

[FSV92] L. Ferreira, M. Sinderen, and C.A. Vissers. Protocol design and implementation using formal methods. *The Computer Journal*, 35(5):478–491, 1992.

[FTW90] N. Fuchs, K.L. Tse, and R. Whitty. Cost management with metrics of specification: the COSMOS project. In *Approving Software Products, editor W. Ehrenberger*, pages 275–278, 1990.

[FW86] N.E. Fenton and R.W. Whitty. Axiomatic approach to software metrification through program decomposition. *The Computer Journal*, 29(4):329–339, 1986.

[Gil77] T. Gilb. *Software Metrics.* Winthrop Publishers, London, 1977.

[Gri81] D. Gries. *The Science of Programming.* Springer-Verlag, Berlin, 1981.

[Hal77] M H Halstead. *Elements of Software Science.* Elsevier, 1977.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, 1988.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa81] C.A.R. Hoare. A calculus of total correctness for communicating processes. *Science of Computer Programming*, 1(1):49–72, 1981.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, London, 1985.

[Hol] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, New York, 1991.

[Hol95] G.J. Holzmann. *Basic Spin Manual.* AT&T Bell Labs, N.J., 2.0 edition, January 1995.

[Hum90] W.S. Humphrey. *Managing the Software Process.* Addison Wesley, 1990.

[IS89] D.C. Ince and M.J. Shepperd. An empirical and theoretical analysis of an information flow based design metric. In *Proceedings of the European Software Engineering Conference, Warwick, UK*, pages 11–16, 1989.

[ISO91] ISO/IEC. Joint technical committee: Information technology - software product evaluation - quality characteristics and guidelines for their use. International standard, ISO/IEC, 1991.

[Jac97] J. Jacky. *The Way of Z: Practical Programming with Formal Methods.* Cambridge University Press, Cambridge, 1997.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Hemel Hempstead, 1986.

[Kin77] J.C. King. *Software Metrics*. Winthrop Publishers, London, 1977.

[KPF95] B.A. Kitchenham, S.L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995.

[KvEvS90] P. Kremer, P. van Eijk, and M. van Sinderen. On the use of specification styles for automated protocol implementation from LOTOS to C. In *Proceedings Protocol Specification, Testing and Verification, editors L. Logrippo, R.L. Probert, H. Ural*, Amsterdam, 1990. North-Holland.

[KWD86] B.A. Kitchenham, J.D. Walker, and I. Domville. Test specification and quality management - design of a QMS sub-system for quality requirements specification. Project Deliverable A27, Alvey Project SE/031, 1986.

[LA90] P.A. Lee and T. Anderson. *Fault-Tolerance: Principles and Practice, Second Edition*. Springer-Verlag, New York, 1990.

[Lap95] J-C. Laprie. Dependability of computer systems: Concepts, limits, improvements. In *ISSRE'95, Proceedings of the Fifth International Symposium on Software Reliability Engineering, Toulouse, France*, pages 2–11, 1995.

[Lor93] M. Lorenz. *Object-oriented Software Development: A Practical Guide*. Prentice Hall, 1993.

[Low91] G. Lowe. Probabilities and priorities in timed CSP. Technical report, Oxford University, 1991.

[LPP70] D.C. Luckham, D.M.R. Park, and M.S. Paterson. On formalised computer programs. *Journal of Computer and System Sciences*, (4):220–249, 1970.

[McC76] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Hemel Hempstead, 1989.

[MMM97] R. Mili, A. Mili, and R.T. Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Transactions on Software Engineering*, 23(7):445–460, 1997.

[Mye79] G. Myers. *Program Testing*. Wiley, New York, 1979.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, (6):319–340, 1976.

[O'N92] G. O'Neill. Automatic translation of VDM specification into Standard ML programs. *The Computer Journal*, 35(6):623–624, 1992.

[PM87] R. Pountain and D. May. *A Tutorial Introduction to Occam Programming*. McGraw-Hill, Oxford, 1987.

[Pra84] R.E. Prather. An axiomatic theory of software complexity measure. *The Computer Journal*, 27(4):340–347, 1984.

[Ros94] A.W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of CAR Hoare*, pages 353–378. Prentice-Hall, Hemel Hempstead, 1994.

[SAH+00] J. Staunstrup, H. Anderson, H. Hulgaard, J. Lind-Nielson, K. Larsen, G. Behrmann, K. Kristofferson, A. Skou, H. Leeberg, and N. Theilgaard. Practical verification of embedded software. *IEEE Computer*, 33(5):68–75, 2000.

[SC01] M. Shepperd and M. Cartwright. Predicting with sparse data. *IEEE Transactions on Software Engineering*, 27(11):987–998, 2001.

[Sch90] S. Schneider. *Correctness and Communication in Real-Time Systems.* Ph.D. dissertation, Oxford University, Balliol College, March 1990.

[Sch92] N.F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, 1992.

[SCMC94] P. Strain-Clark, P. Mett, and D. Crowe. *Specification and Design of Concurrent Systems.* McGraw-Hill, London, 1994.

[SDN+89] W. Samson, P. Dugard, D. Nevill, P. Oldfield, and A. Smith. The relationship between specification and implementation metrics. In *Measurement for Software Control and Assurance, editors B. Kitchenham and B. Littlewood*, pages 335–384. London, 1989.

[She95] M. Shepperd. *Foundations of Software Measurement.* Prentice Hall, London, 1995.

[SK01] M. Shepperd and G. Kagoda. Comparing software prediction techniques using simulation. *IEEE Transactions on Software Engineering*, 27(11):1014–1022, 2001.

[Som98] I. Sommerville. *Software Engineering, Fifth Edition.* Addison-Wesley, Wokingham, England, 1998.

[Spi88] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, Cambridge, 1988.

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Wokingham, England, 1994.

[Tur86] D. Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, 1986.

[US87] D. Ungar and R.B. Smith. SELF: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–242, 1987.

[VPP95] L. Votta, A. Porter, and D. Perry. Experimental software engineering: A report on the state of the art. In *Proceedings of the 17th International Conference on Software Engineering (ICSE'95), Seattle, Washington, USA*, page (no page numbers provided), 1995.

[WB95] H. Waeselynck and J-L. Boulanger. The role of testing in the B formal development process. In *ISSRE'95, Proceedings of the Fifth International Symposium on Software Reliability Engineering, Toulouse, France*, pages 58–67, 1995.

[Whi90] R. Whitty. Structural metrics for Z specifications. In *Z User Workshop '89, Oxford, editor J.E. Nicholls*, pages 186–191, 1990.

[WM90] J.C.P. Woodcock and C.C. Morgan. Refinement of state-based concurrent systems. In *Proceedings of VDM Symposium, Kiel, Germany*, pages 340–351, 1990.

[YW78] B.H. Yin and J.W. Winchester. The establishment and use of measures to evaluate the quality of software designs. In *Proceedings of the ACM Software Quality Workshop*, pages 510–518. ACM, 1978.