A Category Theoretic Approach to Inconsistencies in Modular System Specification

Catherine Menon
A Thesis Submitted in Partial Fulfillment
of the Requirements For the degree of
Doctor of Philosophy
School of Computer Science
The University of Adelaide

April 2006

I would like to thank my supervisors Professor Michael Johnson, Dr Charles Lakos and Dr Sylvan Elhay for their help, guidance and mentoring throughout my studies. Without their encouragement and commitment, this work would not have been possible. I would also like to thank my family and friends for their support and consideration.

This thesis is dedicated to my parents Nanda Menon and Alexandra Menon, and my brother Anand Menon.

Contents

Abstract

Inconsistency management is fundamental to effective system specification. This thesis presents a categorical approach to the detection, classification and resolution of those inconsistencies which commonly arise in modular systems. This categorical approach relies upon the generation of the *Categorical Consistency Framework*, within which inconsistencies can be represented separately from implementation details. Using this framework, we construct a taxonomy of inconsistencies, presenting metrics for determining their severity and discussing a number of resolution methods. We also show how this framework can be used to combine multiple specifications into a single consistent system and predict the behaviour of this system. In the process, we adapt existing work relating to the database view update problem to system specification. This enables us to analyse the probable underlying system behaviour, given the behaviour of individual components. We also introduce the notion of *degrees* of consistency, a concept which allows us to examine how well a system recovers from system failure.

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Signature of candidate

Chapter 1

Introduction

1.1 Introduction

Inconsistencies in a system specification are one of the primary causes of costly delays and specification revisions. An inconsistency is a conflict in the system specification — in the most general sense, a situation "where a relationship which is supposed to hold between two pieces of information does not hold" [21]. Inconsistencies are arguably more common in large modular systems than smaller monolithic ones. This is partly due to the fact that large, modular systems offer more functionality and are therefore more complex. This greater complexity results in more opportunities for inconsistencies to arise when specifying the system. There are several reasons for this relationship between complexity and inconsistencies, most of which involve miscommunication during the specification process.

Firstly, a customer might not communicate his goals for the system clearly to a designer. Naturally, a complex system has more goals — that is, more functionality -- than a relatively simple one, and therefore the risk of a customer failing to communicate every goal is greater. In this case, a designer might build the system exactly to the specification he was given, yet still produce a system which does not satisfy the customer. Clearly, eliciting the goals is a fundamental step of the specification process.

Secondly, even if the designer does understand the goals adequately, complex modular systems often involve multiple customers with multiple goals. Each customer is only concerned about the particular module which will benefit him. For example, given a vending machine, a user of this machine is only concerned that it should deliver the correct product upon request (the delivery module). In contrast, the supplier which stocks the machine will only be concerned about receiving a call when the machine is empty (the notification module). The owner of the machine, however, is interested in *both* of these modules. If the customers have not communicated with each other about their goals, it is possible that the goals conflict. For example, the customer may require that the vending machine contain at least one sample of every product. By contrast, the supplier may only request a notification when at least half the machine needs re-stocking, to save on delivery costs. In this case a designer cannot guarantee to satisfy both these requests due to the inherent conflict, or inconsistency.

Finally, when a specification undergoes any refinement there is the risk of introducing inconsistencies. By definition, refinements alter some of the properties of the system. Without a means of indicating which properties should be immutable, this process can introduce unexpected conflicts by changing properties which should not have been altered. The more complex a system, the more difficult it is to provide a comprehensive list of the effects of changing any particular property. As the original properties of the system mutate, designers can easily produce conflicting versions of the one system. Equally, a designer

might include certain constraints which preclude any further refinement of the system, leading to errors later in the design cycle.

Just as with any programming error, inconsistencies can be resolved more easily if they are detected early. However, detecting inconsistencies is a non-trivial task. The presence of multiple designers, while resulting in a faster specification process, increases the likelihood that inconsistencies go undetected. Again, this is due to miscommunication. For example, if a single designer were to specify the vending machine, he would probably identify the conflict between the goals of the user and the goals of the supplier. However, if one designer specifies the delivery module while another specifies the notification module, then the conflict will only be detected if the two designers communicate adequately with each other about what they are doing. There are several reasons why this might not occur.

Firstly, the goals that the designers are given might be very informal. In this case, even if they compare the requests they received from their customers, the conflict would not necessarily be apparent. Secondly, two designers might use different techniques to specify and develop their modules. To check whether the constraints which one module places on a piece of information match the constraints placed on this same information by a second module, a designer must first translate the constraints into a language common to both modules. This process is time-consuming, and can cause information to be lost. Furthermore, in a modular system, designers may be concentrating on different stages of development. A designer refining the functionality of a completed product is likely to make assumptions about the components which may not be satisfied by an incomplete system.

One way to ensure that an inconsistency is detected is to use some form of formal specification technique. We can then detect when an inconsistency occurs, by identifying those logical rules which are broken. For example, the presence of an inconsistency in a system using first order logic can be detected by the ability to conclude `A AND NOT-A` from this system. A formal specification technique not only helps designers identify inconsistencies in their specifications, but allows the comparison of modules specified at different times, by different techniques, and by different designers. This means that those inconsistencies which only occur when modules are placed together can be detected before the modules are implemented.

However, a formal specification technique alone is not enough. This is because, even if multiple designers use a similar specification technique, the assumptions they make about the system may still be different. For example, the designer specifying the delivery module of a vending machine may assume there is to be no non-determinism in the system. This is because a user, on pressing the buttons for a product, does not want the machine to non-deterministically

dispense any product currently in stock. A designer specifying the notification module (for re-stocking the machine), on the other hand, may assume the presence of non-determinism in the system; this means that the re-stocking notification can be sent to any driver belonging to the supplier company. These choices may then have ramifications in terms of what can be assumed when the two modules are combined.

One way to avoid the problem of designers using different assumptions is to define a mathematical 'meta-specification' framework. This framework should provide a mathematical foundation which enables a designer to analyse all specifications written in a particular language, or class of languages. That is, this framework should be able to be used to analyse specifications regardless of the assumptions of the designer. For example, we should be able to express both deterministic and non-deterministic specifications within the same framework. As another example, if one system requires input from a user while another is completely self-contained, the one mathematical foundation should still suffice for both systems. One reason this is so important is due to the nature of modular systems. In these, a fully-specified system may be used as a module in a wider environment. If the mathematical framework can only analyse one style of specification, then it is difficult to predict the effects of including a module specified in a different style.

The process of including a fully-specified system as a module in a wider environment gives rise to another requirement of any mathematical analysis framework. This is the requirement that, using such a framework, a designer can formally deduce information about the system behaviours from observing the behaviours of a single module. This means that an inconsistency which is caused by an error in a single module can be identified, and the behaviour of that module altered to resolve this. Furthermore, a designer will then be able to identify those subsystems which are essential to the correct functioning of the wider system. This can lead to a change in the direction of specification; for example, an inconsistency which affects these essential subsystems will now be seen to be more serious than an inconsistency which does not. However, in order to express this precisely, the framework needs to provide a means of expressing relationships between a component and a wider system.

1.2 Scope of this Thesis

Within this thesis, we will present a mathematical framework to aid analysis of modular systems. The framework will satisfy the following criteria

- The framework should not be dependant upon implementation details. For example, the choice of variable names should not affect the underlying framework.

- The framework should enable us to detect different types of inconsistencies.
- These inconsistencies should be able to be grouped according to their shared characteristics as observed in this framework. This will aid the search for resolution techniques.
- The one framework should be capable of analysing systems which use different specification styles from each other. These styles might differ in the areas of user input, non-determinism, and the precise definition of a system state.
- Finally, this framework should permit us to draw conclusions about the underlying system, assuming we can observe only selected components.

These criteria have arisen out of the discussion summarised above as being the most useful for a framework used to specify state-based modular systems.

The framework we present in this thesis uses category theory to achieve these analysis goals. Specifically, we show how to represent a system specification by a number of categories and categorical constructions. The categories vary according to the system in question, showing that this technique is not confined to those systems displaying a narrow range of properties. The categorical constructions we define will enable us to examine inconsistencies which occur in component-based systems. By modelling these inconsistencies in the language of category theory, we can identify the changes which need to be made to a system specification, without requiring any specialised knowledge of the specification techniques and assumptions.

In addition, we show how the concept of a *minimal behaviour* can be used in system specification. A minimal behaviour is the fastest, or "least disruptive" system behaviour which permits us to make certain observations on a subsystem. There are a number of different metrics for the speed or disruptive ability of a behaviour, leading to different characterisations of this minimal behaviour property. Notwithstanding these different characterisations, a minimal behaviour can also be used to identify how badly a system is affected by inconsistencies. This is done by first identifying those subsystems whose correctness is essential to the main functionality of the system. We then identify the minimal behaviours which must take place in the underlying system in order that the subsystem contains no inconsistencies.

1.3 Thesis Structure

In Chapter 2 we review the literature relevant to this thesis. This includes existing work in the areas of system specification, mathematical abstractions and requirements engineering. We also examine the language Rosetta [1], which we

will be using to illustrate all examples within this thesis. Chapter 3 shows how the category of theories can be used as a semantic basis for Rosetta and similar languages. This supplies the groundwork for concepts which we will later use in different settings, such as the procedure for combining components. Chapter 4 describes a way to create categories representing a system specification, by making use of EA sketches. We show how these sketches allow us to express states of the system as functors between categories. Based upon the resulting constructions, we then present a taxonomy of inconsistencies. This includes a description of how each class of inconsistency appears within the categorical framework, and a discussion of how different types of inconsistency can be resolved.

In Chapter 5 we demonstrate how this categorical framework can be used to represent a variety of different systems. We also show how to use the framework for analysis beyond that required to identify inconsistencies. For example, we can use these categories to identify the effects of removing or replacing a particular component. The notion of minimal behaviours is also introduced in this chapter. Here, a minimal behaviour in a system represents the fastest way to achieve certain observations on a subsystem. Finally, we compare our categorical framework to others which are commonly used to represent systems and solve problems.

Chapter 6 then discusses how to achieve degrees, or minimum levels, of consistency. This demonstrates the degree of *failure tolerance* of a system, illustrated with the aid of a case study. The degrees of failure tolerance determines how well a system recovers from an external system failure in which data is lost. The different degrees of failure tolerance are also used to define how badly a particular subsystem of the case study is affected by different types of failure. We also show how guaranteeing a minimum level of consistency can aid the refinement process. In particular, we show how we can use the framework developed within this thesis to ensure that essential consistency properties persist throughout the refinement of a system. Finally, in this case study, we discuss the occurrance of the frame problem, or how to specify what properties should *not* change during a particular behaviour. Again, the suggested solutions for the frame problem rely on the notion of degrees of consistency.

1.4 Contribution of this work

This thesis demonstrates how a framework using category theory can be constructured to provide insight into system specification. We generate a single framework within which different types of analysis, including inconsistency management, can be performed. Moreover, systems can be represented in this framework even when they contain inconsistencies. As a result, we can formally compare systems which contain different types of inconsistency. Another practical

implication of this ability to represent inconsistent systems is the added freedom to permit potentially inconsistent refinements. By removing the emphasis on ideal and consistent systems, this framework can be used to consider the practical as well as theoretical sides of system specification.

Within this thesis we also show how existing frameworks and techniques can be applied to system specification. For example, Chapter 4 takes the existing notion of EA sketches and database view updates and applies these to specification languages. This is a new way of looking at view functors defining state, which leads to a new definition of the category of states of a system. In existing theory, natural transformations serve as the transitions from one database state to another. However, when we apply this theory to specification languages, we require a more flexible definition for transitions. This is because the distinguishing characteristics of a state transition vary depending upon those properties of the system.

This thesis also demonstrates how notions of propagatable view updates can be applied to specification languages. This enables existing work on databases to be applied to general specification systems. We examine what happens when certain categorical requirements, such as those for the existence of a pre-cocartesian lifting, are relaxed. This enables us to create new conditions for minimal behaviour which suit system specifications, as opposed to databases. Finally, we show how the results obtained from analysing systems using this categorical framework are analogous to the results obtained using other, established techniques.

The framework introduced by this thesis is able to concisely express the level of consistency required in a system. This means we can formalise the notion of failure transparency, and classify systems based on their robustness. This allows a designer to identify important behaviours and subsystems, formally treating inconsistencies within these as relatively serious. This receptiveness to designer input ensures that, unlike many static analysis frameworks, the framework presented in this thesis can be customised to suit a changing system.

Chapter 2

Background

2.1 System Specification Techniques

Systems can be specified by using a variety of different methods, and the choice of specification method depends upon the size, purpose and intended longevity of the system. System specification techniques may be classified as either informal, formal or semi-formal [93]. Informal techniques include English (or other)-language specifications, and are generally discouraged for complex or large systems. By contrast, formal techniques are mathematically precise ways of encapsulating the system requirements and desired behaviours. We explore these further in Section 2.3.

Possibly the most common system specification techniques are the semi-formal techniques, which include all variants on *structured analysis* [85]. Structured analysis was introduced in the 1960s, and is founded upon an examination of the flow of data within systems. The advantage of structured analysis is that a hierarchy of requirements and components can easily be obtained from data-flow diagrams or equivalent representations. However, the focus of this technique is on the structure of each component in a system, rather than upon the behaviour of components. That is, structured analysis lacks the facility to easily describe the dynamic behaviour of real-time systems. In Chapter 3, we show how an analysis framework using the category of theories and based upon examining data interaction also demonstrates the strengths and weaknesses of structured analysis. Some variants, such as Harel statecharts [42], finite-state automata [13] and Petri nets [79] have been introduced to remedy these deficiencies. In Chapter 4 we show how category theory can also be used for this purpose, demonstrating specifically how it supplies the theories of Chapter 3 with the capability to comprehensively describe state transitions. It is this category theoretical treatment which will form the basis of the framework we proposed in Chapter 1.

In general, semi-formal techniques are constructured from a formal basis, but their notation and semantics are not necessarily formal. Such techniques enable a designer to express concepts precisely, while still retaining a structure which can be understood by the customer. One of the most important examples of a semi-formal technique is the use of *software specification languages.* These are languages designed to represent the constraints and requirements of a system. Specification languages can vary from the hardware descriptive languages used for circuit design like VHDL, to the abstract data-flow diagrams of Harel statecharts. They are commonly used in conjunction with mathematical formalisms such as temporal logic [78] or theorem provers. One of the aims of this thesis is to demonstrate such a combination by using a specification language in conjunction with the mathematical formalism of category theory. The following section considers some of the relevant properties of specification languages, to which we will refer later.

2.2 Specification languages

The majority of examples throughout this thesis will be in Rosetta [1], a specification language under development at Kansas University. However, Rosetta is just one member of a family of languages which we will be examining. Most of the members of this family meet the criteria for the family of languages specified by CoFI [75], the Common Framework Initiative. This is a framework which was conceived in 1995 and intended to form a basis for the development of algebraic specification. The languages within this family include Clear [34], OBJ3 [37], LOTOS [10] and ABEL [18]. In general, these share common properties of modularity, and have similar communication semantics. They can also be described with an algebraic specification technique, such as the technique we present in Chapter 3 for Rosetta.

2.2.1 Common Properties of Specification Languages

The members of the family of languages specified by CoFI share certain properties, due to their common classification as restrictions or extensions of CL, the basic language of CoFI. One common characteristic is that they offer appropriate functionality for algebraically specifying requirements and state-based program design. In this section, we present some of the other characteristics of these languages that we will make use of later in the thesis.

These languages are all modular specification languages, meaning that they partition a system into several modules or components. This can be done *behaviourally* or *structurally*. In a behavioural specification each module within the specification describes a different function of the system. By contrast, in a structural specification each module describes a different physical component within the system. In Section 4.7 we show how these differences can have an effect on the types of inconsistency which arise within a given system. In addition, the majority of these specification languages are capable of implementing the concept of state and state change. In state-based specifications, the values within the system are constrained relative to state. This leads to the fundamental consistency management requirement that the behaviours of components must synchronise [20]. That is, the components must undergo state transitions in such a way that every system constraint is simultaneously satisfied.

Simultaneously satisfying all constraints can be seen as a scheduling problem, whereby the state transitions of components are scheduled to avoid causing conflicts. We will return to this question of scheduling state transitions when considering what makes a behaviour consistent, in Chapter 4. When scheduling system state changes, we also consider the question of stuttering. Stuttering has been defined as "a semantic transition in an execution [which] has no observable effect" [91]. Much system analysis, especially with the family of languages specified by CoFI, considers the number of stutters at any point irrelevant. This is known as a stuttering-insensitive approach, and will be used throughout

this work. Additionally, the systems we examine in this work will generally not be those which demonstrate final behaviours. That is, we will examine systems in which it is possible to observe a behaviour consisting of an infinite number of distinct states.

In order for two modules to interact, there must be some means of communication between them. The languages under discussion achieve this either through the use of shared data or via message passing. Typically any shared data consists of definitions of datatypes and functions, and provides a vocabulary with which modules can communicate. It is this communication which can cause inconsistencies, most notably when shared data is not interpreted in the same way by all components. To address this, the concept of a *data universe* was introduced. Goguen [38] defines a data universe as a fixed data algebra representing the information shared throughout the system. We return to this concept in both Chapter 3 and 4, in which we show how a data universe can be defined and presented within our categorical framework.

The data universe is also used to enforce the notion of data protection, or encapsulation. This is achieved by meta-constraints which prevent modules from constraining elements of the data universe. More strictly, the idea of data protection and encapsulation introduced in [38] states that a component cannot declare functions or constraints which operate solely upon the data universe. Instead, these functions or constraints must be declared as part of the data universe from the start. This ensures that all components have a consistent idea of what the universe should look like, both in terms of its functions and its constraints. In Chapter 3 we discuss some possible customization to the concept of a data universe, making use of Rosetta as a case study.

Finally, the languages for which we design the categorical framework of this thesis all offer a means of combining modules to produce a new module. OBJ3 [37], for example, uses parameterization of interfaces to instantiate new components. In Rosetta, as explained in Appendix A, it is possible to create new components by pre-determined methods known as *interactions*. Consistency issues arise here also, as the behaviour of the new component is dependent upon the behaviour of the original components. In order to analyse these complexities, we often turn to particular inconsistency management techniques used in conjunction with mathematical formalisms. Section 2.5 introduces the common inconsistency management techniques, some of which we apply using our categorical framework in Section 4.7. Because this creation of a new module from existing specifications is a fundamental tool in system design, we will ensure that our categorical framework provides a means of explicitly representing this. This is covered in more detail in Chapter 3.

This work was motivated by a need to provide a formal semantics for Rosetta. While this is no longer the stated objective of the research, it serves as a source

for many of the examples. Similarly, some of the semantic discussions, while generally applicable, use Rosetta-specific terms and concepts. Because of this, we describe here Rosetta and one of the languages (Clear [34]) which had the most influence upon its development. Appendix A contains more information on the syntax and semantics of Rosetta.

2.2.2 Clear

Clear [34], a language designed by Goguen and Burstall in the 1970s, makes use of a mathematical foundation using *theories*. Under the semantics of Clear, models of these theories are defined to be *algebras*. We discuss theories and algebras in more detail in Sections 2.3.3 and 2.3.1, and use these concepts in Chapter 3 when discussing how to formally combine components. That is, we will retain some properties of the semantic basis of Clear, and extend these to form a semantic basis for more complex languages such as Rosetta. Informally, theories are a means of abstracting away from the syntactic details of a specification. They also allow renaming, which permits the identification of similarities between specifications which are syntactically different.

The semantics of Clear use categorical constructions to describe the composition of two or more components. These constructions, detailed in [32], include colimit diagrams, comma categories and two-dimensional categories. These are used to model a number of different operations such as the derivation of new theories from existing ones, abstracting or hiding theories and inductively completing theories. In Chapter 3, we show how applying similar techniques to the category of theories enables us to describe the combination of components within a system. The parameterized modules of Clear have formed the basis for other powerful algebraic specification languages, such as OBJ3 [37]. Less directly, the modular nature of Ada [30], ML [17] and LOTOS are attributable to the examples of Clear.

2.2.3 Rosetta

Rosetta [1] is a specification and requirements modelling language currently under development at Kansas University. The purpose of Rosetta is to support concurrent systems level design, a similar aim to that of VHDL [2] and other hardware descriptive languages. However, Rosetta was developed due to a perceived lack in the semantics of these languages. Specifically, relatively few such languages are provided with a formal semantics comparable to the semantics of Clear. Rosetta seeks to address this deficiency by providing the same level of detail when modelling systems as these hardware descriptive languages, but with a formal semantic basis and an increased attention to specifying the requirements of a system. The modular nature of the systems specified in Rosetta mean that the semantics presented in this work will be based upon the concepts introduced by Clear. Throughout this thesis Rosetta serves as a case study, allowing us to demonstrate how a combination of a formalism such as category

theory and a specification language enables us to describe the composition and consistency of a system.

Rosetta allows a system to be specified from multiple perspectives by adopting a syntax which defines *domains* of activity. These pre-determined domains provide a vocabulary for undertaking specific tasks. For example, one domain is used for mechanical engineering specifications, one for trace-based analysis, and so on. The advantage of using the categorical framework we present in this thesis to unite these domains is that it enables components to easily communicate with each other, even when their proposed functions are vastly different. One of Rosetta's main aims is to describe the way a system can be composed from these individual components. This naturally leads to questions of consistency and synchronisation of shared information, a topic which we shall return to in later chapters.

Since Rosetta is a language under development, the syntax and semantics are in flux. Appendix A contains a description of some of the fundamental properties of Rosetta, as well as a subset of the functions and syntax. However, we make no claim that the work we present here necessarily provides a comprehensive semantics for the entire language as it stands. Multiple semantic bases have already been proposed for Rosetta, including the coalgebraic discussions presented in [64], the theory-based semantics of [73] and the categorical analysis of [74]. It is to be hoped that this work will further the search for a comprehensive formal semantics, and will serve as an illustration of how category theory can be used to formalise a general specification language.

2.3 Formalisms and Mathematical Abstractions

While specification languages are ideal for specifying the behaviour of a particular system, they do have some drawbacks. Firstly, it is often difficult to determine the relationship between two specifications written in different languages. While it is possible to circumvent this problem using translators (such as XEludo [67], which translates LOTOS to Prolog), these tools must necessarily be language-specific. Furthermore, most inconsistency management techniques require knowledge of the language in question. This means that when analysing a system to detect and resolve any inconsistencies, a more formal basis is needed to perform this analysis in a thorough manner.

Mathematical specification techniques, or *formal methods*, can be used to abstract completely away from these syntactic and language-specific issues. The broad term "formal methods" refers to the use of mathematical techniques to design and analyse systems [86]. Furthermore, the notation used for these techniques must have a complete formal semantics [44], which enables us to discuss the correctness of a specification. The advantage of having a complete formal semantics for notation means that inconsistencies occur in identifiable and

provable ways. For example, in a system of first-order logic an inconsistency presents as an axiom `A AND NOT-A`. However, "formal methods" is a broad classification, and we will mainly restrict our attention to the most common and mathematically rich specification techniques or aids. These include equational logic, algebraic theories, category theory, and coalgebras.

More specifically, the majority of the languages to which this thesis applies are amenable to algebraic specification techniques. Algebraic specifications of abstract datatypes were proposed by Ehrig in the 1970s, and were restricted to many-sorted algebras and equational axioms [46]. Nowadays this approach has branched out to include partial functions and complex logical systems [14]. Because of the importance of algebraic specification to Rosetta and similar languages, we present a brief introduction to the fundamental concepts of a many-sorted algebra. In Chapter 3 we will use these definitions to formalise the semantics of Rosetta, and show how they can be used to precisely describe the relationships between components within a system.

2.3.1 Algebras

A many-sorted algebra A is associated with a signature $\Sigma = \langle S, \Delta \rangle$. Here, S is a set known as the set of *sorts*, and will eventually be used to represent the datatypes in a system. Δ is a set of operator names, where the domain and range of each operator are elements of S^* and S respectively. Δ will eventually be used to represent the functions in the system. As is standard, S^* here represents the Kleene-closure of S. An algebra A associated with a signature Σ is often termed a Σ-algebra, or model of Σ, and is defined by a denotational function γ_A. This function associates each sort in S with a set (these sets are known as the *carriers*), and each operator name in Δ with a function of corresponding rank and arity. For each signature Σ, the Σ-algebras form a category. Morphisms within this category are Σ-homomorphisms, as presented in [31]. These morphisms preserve the structure of algebras by preserving the actions of each operator within Δ.

We can also require that the functions of A obey a family Υ of equations or axioms. Together with the signature, these axioms form the *presentation* $P = \langle \Sigma, \Upsilon \rangle$. A Σ-algebra in which the functions obey the axioms of Υ is said to *satisfy* the presentation. and the category of these Σ-algebras is known as the *variety* over P. Each algebra in the variety corresponds to a different way of interpreting P. For example, the *initial algebra* [36] over P possesses the no-confusion and no-junk properties. The no-confusion property states that two functions in Δ should only be equal in the algebra if there is an axiom in the presentation P which enforces this. The no-junk property states that there should be no element of any carrier set which cannot be obtained by an application of an operator in Δ.

Initial algebras have traditionally been of interest because they exclude from consideration a number of trivial models of a presentation. Another means of eliminating these trivial or degenerate models is by the use of inequations [68], or necessary inequality operators [92]. These operators denote the data elements which should not be equal in any model (algebra). Yet another common technique for eliminating certain trivial models is by the use of constraints designating that a part of the specification should be implemented as an initial algebra. That is, this part of the specification should possess the no-junk and no-confusion properties. These constraints have been used in algebraic specification languages such as Clear [34] and ABEL [18] to denote constructive specifications of datatypes. We will use this idea of a constructive specification in Chapter 4, where we show how this 'initiality' can be obtained by categorical constructions. This ability is important in terms of inconsistency management, since an inconsistency may arise when two components do not consider the same data elements to be equal.

An algebra can therefore be seen to represent a simple system specification in terms of sets and functions. While the above definition allows for specification of very simple systems only, later developments in algebras permit more sophisticated reasoning and datatypes. In addition, tools such as theorem provers for algebraic specifications have been constructed to aid analysis. These include the optimising compiler [41] for OBJ3 and the proof support tool InterACT [63].

2.3.2 Meta-formalisms

While algebras provide a useful semantic basis for some types of specification, analysis which is restricted to the use of algebras as introduced in Section 2.3.1 is often limited in scope. For example, it is not obvious how two algebras which satisfy different presentations might be related. Similarly, in order to generate a new algebra from existing ones — a problem which might arise when combining components within a single system — we must use the added vocabulary and constructions obtained from category theory. Finally, not every specification can be easily represented using algebras. For example, a number of different semantic bases exist for Rosetta alone, including an algebraic basis, a coalgebraic basis and a number of less formal approaches. Each of these is suited to a different type of specification.

This proliferation of different formal methods used for the semantic basis of Rosetta illustrates a more general problem with system analysis. Namely, a specification in one logical system cannot necessarily be rigorously compared to a specification in another system, just as a specification in one specification language may not be easily or automatically translated into another. Additionally, a tool designed for one formalism is not guaranteed to work in the context of another. This means that tools will not offer the same functionality when the language (either formal or informal) of a specification changes, as often occurs during system refinement.

Since one of the aims of this thesis is to provide a means of identifying inconsistencies when a number of potentially very different components are combined, we will require a means of unifying the different formal methods which might have been used to specify each component. This can be achieved by the use of a *meta-formalism*, a concept introduced in the 1980s to unify different formalisms. The best-known example of a meta-formalism is an *institution*, introduced by Goguen [31].

2.3.3 Institutions

Institutions provide a framework for discussing system specifications and the models which satisfy them. Different formalisms may have differing notions of models and of satisfaction, which makes it difficult to relate a model in one formalism to a model in another. We can negate this problem by associating each formalism with an institution. Institution morphisms can be used to relate one such institution to another, and therefore relate models of one formalism to models of another. These morphisms were used to show [31] that under certain conditions a theorem prover for one logical system can be used for another logical system. Institutions are the best-known and most general meta-formalism, but there exist several variants, such as a logical framework [43], or Π–institutions [26]. Institutions originally arose out of the semantics of Clear [34], although they have been used in a variety of contexts, from databases to artificial intelligence.

Definition 2.1. *An* institution *[31] consists of:*

- *A category* $\boldsymbol{Sign}$*, with objects known as signatures and morphisms known as signature morphisms*
- *A functor* $Mod : \boldsymbol{Sign} \rightarrow \boldsymbol{Cat}^{op}$*, associating each signature* Σ *with a category whose objects are known as* Σ*-models, and whose morphisms are known as* Σ*-model morphisms*
- *A functor* $Ax : \boldsymbol{Sign} \rightarrow \boldsymbol{Set}$*, associating each signature* Σ *with a set whose elements are known as* Σ*-sentences, or* Σ*-axioms.*
- *A satisfaction relation* $\models_\Sigma$ *relating* Σ*-models and* Σ*-sentences, such that for all signatures* Σ, Σ' *in* $\boldsymbol{Sign}$*, and for all signature morphisms* $\phi : \Sigma \rightarrow \Sigma'$

$$m' \models Ax(\phi)(e) \text{ iff } Mod(\phi)(m') \models e$$

for each object m' *in* $Mod(\Sigma')$ *and each* Σ*-sentence* $e \in Ax(\Sigma)$

One of the fundamental properties of institutions is the inclusion of a satisfaction relationship $\models$ which is consistent under change of notation. In Chapter 3, we use the satisfaction relationship to identify whether differences between components are semantic, or merely syntactic.

Algebras can be interpreted in the context of institutions by examining the *algebraic institution* [31]. We introduce this institution in particular since it will be used in Chapter 3 to illustrate the semantics of Rosetta. While Rosetta semantics can be expressed using algebras alone, the added generality of the algebraic institution means that we can formalise the concept that truth is invariant under change of notation in all Rosetta specifications. In addition, using institutions to provide a semantics for Rosetta allows us to relate the multiple existing semantic approaches for this language to each other.

Within the algebraic institution, **Sign** consists of all algebraic signatures Σ, of the type introduced in Section 2.3.1. Mod takes each signature Σ of this type to the category of $\Sigma-$algebras, also introduced in Section 2.3.1. The functor Ax takes each signature Σ of this type to the set of all axioms which relate elements of this signature to each other. In the vocabulary of Section 2.3.1, the set $Ax(\Sigma)$ consists of those elements which are in Υ for some presentation $P = \langle \Sigma, \Upsilon \rangle$. The satisfaction relationship $\models$ of this institution denotes those algebras which satisfy a particular presentation. Specifically, an algebra m is related (via the satisfaction relationship) to an axiom e if and only if the algebra m satisfies the presentation $\langle \Sigma, e \rangle$.

Theories

Institutions allow components which were originally specified using different formal methods to be combined within a single system. In order to represent a single component within the general framework provided by institutions, we use *theories*. A theory consists of a set of sentences (or axioms) and a signature. Theory morphisms describe the relationship between one theory and another, allowing for the change of signature. A system then consists of the combination of several theories, each representing a module. We demonstrate the creation and analysis of these systems in Chapter 3.

Without a meta-formalism such as that offered by institutions, theories are specific to a certain logical system. For example, a model of a theory may be a coalgebra, an (initial or final) algebra, and so on. While this is acceptable when working within a single logical system, it makes it difficult for designers to compare components specified using different assumptions or systems. The use of institutions enables us to consider theories separately from their models, and therefore compare specifications originally written using different logical frameworks. To aid this discussion in later chapters, we present the following definition of a theory from [31].

Definition 2.2. *If Γ is some institution and Σ a signature in the category of signatures associated with Γ, then a Σ-theory is a pair $\langle \Sigma, E^* \rangle$ where E^* is an equationally closed collection of Σ-sentences.*

In this context, *equational closure* refers to the semantic closure of a set, most commonly used to define a set which is closed under equational logic. A theory

represents the semantic closure of a set of axioms, and therefore describes any entity up to *axiom abstraction* [34] only. We can define the category of theories **Th** associated with the institution Γ as follows.

Definition 2.3. *The category of theories* ***Th*** *defined over the institution* Γ *is comprised of*

- *Objects which are pairs* $\langle \Sigma, E^* \rangle$ *where* Σ *is an object in* ***Sign*** *and* $E^* \subseteq Ax(\Sigma)^*$
- *Morphisms* $\phi : \langle \Sigma, E^* \rangle \rightarrow \langle \Sigma', E'^* \rangle$ *which are those signature morphisms* $\phi : \Sigma \rightarrow \Sigma'$ *such that for every sentence* $e \in E^*$, $\phi(e)$ *is in* E'^*.

Morphisms within the category of theories are referred to as theory morphisms. For later use, we also define a functor *Sign*: **Th** $\rightarrow$ **Sign** with the following action:

- $Sign(\langle \Sigma, E^* \rangle) = \Sigma$
- *Sign* sends ϕ as a theory morphism to ϕ as a signature morphism

This functor allows us to perform syntactic checking on a component specification. It has been shown in [34] that the category of theories contains all finite colimits. This will be used in Chapter 3, when discussing how to generate the theory representing a new component from the theories of existing components. In subsequent discussions we will confine our attention to theories defined over the algebraic institution. When considering this category of theories, the most important theory morphisms will be those morphisms α for which $Sign(\alpha)$ is monic. That is, where the signature of the target theory is completely contained within the signature of the source theory, up to renaming which preserves the distinction between separate elements of the source theory's signature. We term these *signature-injective* theory morphisms, and use them to model refinements or extensions of specifications.

Throughout this thesis, category theory will be used to generalise results obtained within a specific logical framework and, more abstractly, to relate component specifications and system states. The following section introduces some of the categorical constructions which we will be using.

2.4 Category Theory and System Specification

Category theory has been used in many areas of computing since its development in the 1940s [24]. While it has its origins in group theory, Lawvere proposed that it could also be used for the study of logic [65]. This led to the use of category theory in many computing applications, from the theory-based semantics of Clear [34] to the application of colimits in curried functions [8].

Because category theory is not tied to any particular computing language or specification technique, it has also been used to analyse abstract notions such as congruence and bisimulation [29]. In circumstances such as this, category theory is typically used in conjunction with other specification techniques, such as algebraic specification.

Category theory today is still being used in modular system specifications to describe how the modules interact and operate. For example, the EPOXI [25] project uses algebraic specifications and categorical diagrams to describe the structure of modules. Certain constructions and techniques of category theory have proved particularly applicable to computing, notably in the areas of specification and inconsistency management. One such technique is the use of bicategories to model the behaviour of processes. By using simple categorical constructs such as pullbacks, a designer is able to identify behaviours of multiple processes which can coexist in a consistent system. In Section 5.5.2 we show how it is possible to translate between the formal framework we develop in this thesis, and existing frameworks using bicategories. To demonstrate why this is useful, the following section illustrates some of the ways in which bicategories have been used, and some of the benefits they offer for system analysis and inconsistency management.

2.4.1 Bicategories and Processes

Bicategories have a number of uses in applications involving computation. They have been used to model circuit design [60], extended to weak n-categories in theoretical physics [5], and used to model interacting processes [62]. This last has resulted in noticeable practical benefits to computer science. One of these is the development of a technique for compositional modelchecking which reduces some of the impact of the statespace explosion problem. This was demonstrated in [84] with reference to the Dining Philosophers problem. Additionally, the linear-time algorithm for minimizing automata (identified by use of bicategories) presented in [61] applies to an infinite number of similar problems. Because so much existing analysis relies on bicategories, relating our formal framework to bicategories — as we do in Chapter 5 — lends the work presented here validity in a wider context.

The bicategory of spans of graphs, (**Span(Graph)**), has been shown in [62] to describe an algebra of transition systems. Labelled transition systems can be used to represent a system in which components must operate concurrently, which includes synchronising on their input and output where applicable. We describe the bicategory **Span(Graph)** below and show how it can be used to identify those behaviours of different components within a system which must synchronise for reasons of consistency. We will return to this category theoretic approach to inconsistency management in Chapter 5, where we discuss the similarities between **Span(Graph)** and the framework presented in this thesis.

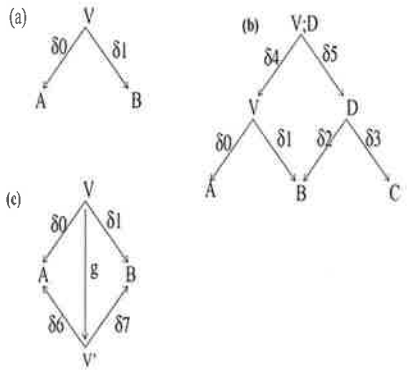

Figure 2.1: (a): A morphism in **Span(Graph)** from A to B; (b) The composition of two morphisms in **Span(Graph)**; (c) The morphism g in **Graph** represents a two-cell in **Span(Graph)**

The category of spans of graphs

Span(Graph) is a bicategory in which the objects (0-cells) are directed graphs and the 1-cells (morphisms) are *spans* of graphs. A span V from graph A to graph B is a pair of arrows δ_0, δ_1 from a common vertex graph V to graphs A and B, as in Figure 2.1 (a). (Horizontal) composition in **Span(Graph)** is defined by using the pullback of two graph morphisms with a shared target, as illustrated by the morphisms δ_4, δ_5 in Figure 2.1 (b). Specifically, the composition of morphisms $V : A \rightarrow B$ and $D : B \rightarrow C$ is the span $V;D : A \rightarrow C$ where the nodes and edges in $V;D$ are labelled according to the following set construction:

$$\{(v,d) | v \text{ is a vertex (edge) of } V,\ d \text{ is a vertex (edge) of } D,\ \delta_1(v) = \delta_2(d)\}$$

Since the pullback is unique up to isomorphism only, we choose one of these to be the 'canonical' pullback. The 2-cells in **Span(Graph)** are defined as being those morphisms which make the triangles commute as in Figure 2.1 (c).

2.4.2 Representing Data and Systems Categorically

Prior to using any categorical constructions (such as the bicategories discussed above) to analyse a system, we must find a way to represent the system elements and their relationships categorically. The category of theories is one possible representation, as pushouts can be used to describe the combination of components within a system. This can be seen in the construction of a semantics for Clear, and in the work we present in Chapter 3. However, the category of theories does not allow us to represent each individual component by a separate category. Instead, it describes the relationships between components by means of morphisms within a single category. To represent each component by

a separate category we will use database analysis techniques involving *entity-attribute sketches* [53]. These sketches allow us to associate a database with a category wherein objects represent entities of the component and morphisms represent relationships between these entities. As we show in Chapter 4, this technique can be adapted in order to associate a category with each component in a system.

Representing each individual component by a separate category has several benefits. Firstly, the constraints on this component can be modelled using category theory. This means that we can analyse inconsistencies using categorical constructions, since all constraints upon a component can be expressed within the category representing this component. Secondly, we can describe the perspective each module has upon the system by means of a *view functor*, or a restriction of the category which represents the entire system. This means that we can easily identify any shared information and potential inconsistencies. Finally, we can identify relationships between two components by examining the relationship that exists between their associated categories. This lets us construct new components from existing ones and relate their behaviours to the original system.

Category theory has a history of being used in database analysis, most notably with the graphical modelling technique of EA sketches [53], to which we have alluded above. Because of the success of this technique in terms of database analysis, we will adapt it within this thesis and apply it to system specification. This adaptation will be introduced in Chapter 4, where we identify an EA sketch which represents the data and constraints within a system or component. Furthermore, in Section 5.3, we will apply some categorical analysis which has been previously used only for databases [54, 55, 56, 57] to a system specification. These later chapters assume a certain familiarity with the basic concepts of EA sketches. To provide a reader with sufficient background, we present here a brief overview of EA sketches and some associated analysis.

2.4.3 EA Sketches: Representing Constraints on Data

Entity-relationship modelling [15] is one of the more common graphical modelling techniques used to construct and analyse databases. In order to provide the resultant representations with more flexibility, category theory was used to express system constraints and database queries [53]. This work was extended in [54], where database updates were represented as spans within a 2-category.

EA (entity-attribute) sketches are a result of this extension. These are a way of representing entities and their relationships, where instances of these entities make up the information in a database. For example, the instances of the entity `Name` in a database might consist of the names of all the people whose records

are stored. EA sketches can therefore be used to represent the relationships between the different fields or records in a database.

Definition 2.4. *An EA Sketch $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ is a finite limit and finite coproduct tuple consisting of:*

- *A directed graph G which is acyclic and finite*
- *A family $\mathcal{D}$ of some of the pairs of paths in G with a common source and target*
- *A set $\mathcal{L}$ of some of the cones [55] within G, including the cone with empty base*
- *A set $\mathcal{C}$ of some of the cocones [55] within G*

In the directed graph G, nodes represent entities and edges represent the relationships between these entities. The family $\mathcal{D}$ contains some extra information about the relationships within the database; each pair in D represents two relationships which are constrained to be equal. For example, one special case is a pair of database queries which produce the same result. Finally, an EA sketch identifies those cones and cocones which are realised, given the constraints of $\mathcal{D}$, as limit cones and colimit cocones.

As shown in [54], each EA sketch presents a category $\mathbf{C}$. This is defined as the free finitely complete category with finite coproducts on G, subject to the images of $\mathcal{D}, \mathcal{L}$ and $\mathcal{C}$ being respectively equal morphisms, limit cones and colimit cocones in $\mathbf{C}$. $\mathbf{C}$ is often termed the *query language*, and is initial in all categories generated from the EA sketch with these restrictions. A model of a database specified using this EA sketches is a functor $D : \mathbf{C} \to \mathbf{Set}$ which preserves finite limits and colimits.

Subsequent papers [55, 56, 57] have built upon this construction, most notably by discussing the characteristics of a database represented by an EA sketch. These works use the category $\mathbf{C}$ to formally identify what makes a view of this database *updatable*, and what algorithms can be used to perform a general update. In later chapters we develop a system specification analog to an updatable database view, and examine what such a view can tell us about the system as a whole. However, to do this thoroughly we will need to know what constitutes an updatable database view, and what can be deduced about a database from this knowledge. We will then be able to relate the conclusions obtained from system specifications to the conclusions obtained from database analysis. The following section discusses some of the issues related to database views, many of which we will return to when applying the same concept to system specification.

2.4.4 System Specification and Database View Updatability

One common question which arises when analysing a database is determining whether a particular view [19] is updatable. An updatable view is one for which any update possible in the view is possible in the underlying database, and the necessary updates to this underlying database can be identified. Identifying updatable views can be very complex, since any given view may not include all entities and may not include all relationships pertaining to those entities which are included. These relationships are what govern the changes which can be made to a database. Because of this, in order to perform a simple update to what we can see in the view it may be necessary to perform a series of more complex updates on the underlying database in order to retain consistency.

The question of updatable views has a system specification analog in terms of the interaction of a subsystem with the larger system constituting its environment. The view of the database corresponds to the subsystem, while the interaction of the subsystem with its environment is governed by the system constraints, which correspond to the relationships between database entities. This correlation suggests, as we show in Chapter 4, that categorical analysis techniques for database view updates can be used to identify system behaviours given knowledge of subsystem behaviours. By deducing potentially conflicting characteristics of the system simply from subsystem observations, we can more easily identify inconsistencies within the system. Here we present some of the aspects of the view update problem and discuss existing works which use categorical techniques to address these issues. We will then use this when examining the system specification analog of the view update problem.

A Categorical Approach to Database View Updatability

In general, an update to a particular view may not be able to be made in the context of the entire system for one of two reasons:

- Due to database constraints relating the visible information to the hidden, there may be no consistent system update which restricts to this requested update in the view.
- There may be several underlying updates which restrict to this requested update, and no indication as to which of these is desired.

The view update problem is then the problem of ascertaining whether a particular update to a view can be achieved in a consistent system, and if so how the update should take place.

Historically, the view update problem has been considered in terms of underlying updates to the database. That is, approaches to this problem have basically consisted of identifying those views for which certain updates are propagatable.

However, this does not lead to a particularly unified approach. For any given view update, there may be many different lists of how this can propagate, and these lists may change as new techniques are found.

The use of EA sketches [56], presents a universal property for view updates. This, in essence, determines both *when* a view update is propagatable (based upon the current state of the database) and *how* the update is to be propagated. A view update is said to be *propagatable* [56] if there is a unique minimal change to the database. We consider this minimal change, which causes the least disruption to the underlying database, to be the *canonical update*. By using a universal criterion for view updatability, the relationship between updatable views and fibrations [49] was identified.

As we have mentioned, we will demonstrate in Section 5.3 how this unified framework of EA sketches and database view updates permits us to draw conclusions about the behaviour of a system from limited information. This will then be extended in Chapter 6 to identify inconsistencies within a system, again assuming we are given only limited information.

2.5 Inconsistency Management

All of these specification and analysis techniques identify inconsistencies in different ways. Some of them provide safeguards against particular types of inconsistency, but not in any comprehensive way. For example, many specification languages contain a syntactic safeguard to prevent certain inconsistencies (such as assigning a value of the wrong type to a variable). However, there inevitably remain inconsistencies which cannot be detected syntactically. This gives rise to the field of *inconsistency management*.

Inconsistencies in a system are broadly defined by Easterbrook as the result of a situation in which parts of a specification do not obey some relationship that should hold between them [21]. In this thesis, we mainly confine our attention to logical inconsistencies. These can arise from incomplete formulation of requirements, lack of care when refining a system, mistakes due to multiple design perspectives and languages, and even errors in the execution of the final correctly-specified system. In a survey paper, Robinson [80] suggests three technical difficulties which are the source of most inconsistencies: voluminous requirements, changing requirements and complexity of requirements. These subsume the more detailed causes above.

In the broadest terms, inconsistencies arise from under-specification or over-specification [21]. Under-specification means that there is not enough information provided. Two designers may then choose different permissible interpretations, which can lead to inconsistencies when these are combined in one system.

Over-specification, on the other hand, can be seen in cases where a variable is over-constrained. This occurs most obviously where multiple specifications overlap [89], and each imposes constraints on the area of overlap. Detecting these type of inconsistencies involves recognising precisely where the overlaps occur. One way to do this is to use the properties of the representation language itself. For example, in set-based languages such as Z [94], set inclusion is a signal that overlap is occurring. Other methods [66, 83] for identifying overlap involve tagging information represented in different components with names from a shared naming hierarchy. A comparison of the tags will then identify shared information.

One common technique for reducing the complexity of requirements is *partitioning* [39]. This is the process of dividing the specification up into its constituent parts, in order that requirements might be managed more easily. One of the best-known usages of partitioning is the Viewpoints method, proposed by Finkelstein [28] and elaborated by Easterbrook [21]. Here, each partition is defined with the aim of detecting inconsistency, and consistency rules must apply to the amalgamation of any two partitions. These consistency rules define whether or not two components may coexist in a system, and must be specified by the designer of the components in question. We return to the question of partitioning in Chapter 4, in which we adapt it to the categorical framework of this thesis.

When an inconsistency is detected, there are a number of different actions which may be taken. The presence of an inconsistency does not itself necessarily indicate that it must be immediately resolved; at a high level of abstraction, permitting inconsistencies enhances the design freedom [27]. Because of this, there are a number of appropriate actions which may be taken when an inconsistency is detected. Firstly, the inconsistency may simply be ignored. This occurs when an inconsistency is isolated, or otherwise has little effect on the system. Alternatively, any action towards resolution may be postponed. In this case the inconsistency is marked, as described in [6], so that its effect on the rest of the system can be traced. In general, this postponement should only occur when there is a reasonable expectation that the inconsistency will resolve itself. If this is not the case, then the inconsistency may be circumvented or ameliorated. These techniques require modification of those constraints which are in conflict. There are a number of different methods for this, including re-structuring the constraints, relaxing constraints, or compromising (such as choosing another value out of an acceptable range for the variable causing conflict) [80]. Finally, if an inconsistency is likely to have a significant detrimental effect on the system as a whole, the best course of action is to immediately resolve it. Again, there are a number of methods for this, ranging from relaxing constraints to invoking external automated tools [81], or requesting input from the designer.

In order to judge the correct response to different inconsistencies, it is necessary to classify them according to their properties and effect upon the system. This involves creating a taxonomy of those inconsistencies which arise in particular classes of systems. One example of this is the classification developed in [88] for compilation errors. In Section 4.7 we present a taxonomy of those errors which occur commonly in system specification, in terms of the categorical framework we develop. Of course, some of the responses to an inconsistency are more involved than others. For example, when immediately resolving an inconsistency, it is necessary to identify those constraints which should be removed or relaxed. This requires some prioritization, discussed in [82] and [59], so that the constraints removed are those which are a factor in the most inconsistencies, or alternatively those which it is cheapest to remove.

A similar technique is used for circumventing or ameliorating the effect of an inconsistency. However, we have mentioned that an isolated inconsistency may be tolerated or ignored. One problem with this is that an inconsistency in a system means, by a strict application of logic, that we can conclude anything from the axioms. To rectify this, quasi-classical logic [47] was developed. This is an adaptation of classical logic which permits reasoning in the presence of inconsistencies. In this way, it negates the effect of the inconsistency on the rest of the specification. It is anticipated that future work using the framework developed in this thesis will be able to include a consideration of this logic adaptation.

2.6 The Frame Problem

Another software engineering concept stemming from issues of consistency is the frame problem [69]. The frame problem refers to the question of how to establish the precise properties which may be altered during any procedure or operation. More specifically, the frame problem is the problem of how to specify that *only* the properties explicitly mentioned may be altered. The axioms which would enforce this are known as frame axioms. In general, identifying and listing the frame axioms for any given operation or event is non-trivial. Furthermore, such lists of axioms do not work when analysing object-oriented languages. This is because the existence of frame axioms would prevent any refinement of an inherited function or method, as discussed in [11].

One way in which the frame problem can be addressed is by using axioms which provide information about what is *absolutely required* to achieve a certain goal. This technique, introduced in [40], allows a designer to say when minimal goals, or minimal behaviours, must be observed in order for a particular event to occur. We expand upon this technique from a categorical perspective in Chapters 5 and 6. Furthermore, we introduce properties of a system which allow frame axioms to be introduced into a system specification, and hold even in the presence of inconsistencies.

Chapter 3

A Structural Formalism for Rosetta

Within this chapter we will show how the category of theories, introduced in Definition 2.3, can be used to provide a formal semantic basis for a general class of specification languages. This class includes many languages which are specified by CoFI [75], and hence are amenable to a semantics based upon algebras. The motivation throughout this chapter will be the eventual attainment of a semantic basis for the specification language Rosetta, an introduction to which is provided in Appendix A. Because of this motivation, the syntax of any examples here will be that of Rosetta, and the semantic discussion will also be biased towards this language. However, the semantics of Rosetta is intended to be merely a motivating example for the demonstration of the utility of a formal framework based on the category of theories. Such a framework should allow us to model the relationships between components in a manner that is not dependent upon the language used to specify the system. Additionally, it should precisely describe the constraints which make up a system specification, and which must be satisfied if the system is to be consistent.

In Section 3.2 we identify two informal but central tenets of Rosetta and related languages. The first of these determines when two specifications are considered equivalent, while the second determines when one specification is said to be an extension of another. These tenets describe the most fundamental relationships between system components, and must be supported by any proposed semantic basis. Section 3.3 proposes such a semantics, based upon the category of theories defined over the algebraic institution. Further to this, Section 3.3.4 describes how these semantics can be used to formalise the different ways components can be combined. Within this section, we also show how the pre-defined Rosetta facet interactions described in Appendix A are supported, and use category theory to describe how designers can create their own facet interactions. This can then be generalised to other specification languages.

3.1 Advantages of Mathematical Formalisms

One of the facilities provided by most specification languages is the ability for designers to combine components into a single system. In Appendix A we introduce some pre-defined methods of combining Rosetta components, known as *facet interactions*. Each interaction describes a different way in which two components in a Rosetta system can be related. The underlying concepts of these interactions are not unique to Rosetta; each describes a means of combining components which is common in systems specified in any language.

Whether in Rosetta or another language, designers require the freedom to create new components from existing ones, using relationships which are not pre-defined. The presence of a formal basis for this type of analysis is essential, in order to ensure that the resultant component possesses those properties that were intended by the designer. A formal foundation also allows us to analyse components and systems using a vocabulary which is not specific to a single

specification language or style. Additionally, by using a mathematical formalism to abstract away from syntax details, we may be able to identify common properties between apparently unrelated specifications. This is because abstractions allow us to identify some underlying properties of systems which may not have been apparent in the original specification language. In this way we ensure that the analysis has validity in a wider theoretical context.

3.2 Central Tenets of Specification Languages

Any formal framework which provides a semantic basis for Rosetta must satisfy two central tenets of the language. These are also fundamental properties of the other system specification languages described by CoFI.

1. Two specifications which differ only by using different variable names or expressing equivalent axioms differently should be recognised as equal specifications. That is, these two specifications will be satisfied by precisely the same two models. For example, the following two Rosetta facets both describe a partial Stack datatype, despite their superficial differences:

```
facet Stack::logic
  push(Stacktype, Datatype): Stacktype;
  pop(Stacktype): Stacktype;
  MyStack: Stacktype;
  MyObj: Datatype;
  begin
    1:pop(push(MyObj, MyStack)) = MyStack;
end StackType

facet ReverseQueue::logic
  add(RevQtype, Datatype): RevQtype;
  rem(RevQtype): RevQtype;
  MyElement: Datatype;
  element: Datatype;
  begin
    1:rem(add(MyElement, MyRQ))=MyRQ;
end ReverseQueue
```

2. If the specification `f2` *extends* the specification `f1` by adding constraints or data elements, then any system satisfying `f2` should also satisfy `f1` within the formal framework. That is, as long as an implementation or model satisfies the constraints of a specification then *regardless of what other behaviours the model displays*, it satisfies the specification. We require that this property be explicitly provided in any formal semantics of these languages. This corresponds in Rosetta to facet extension, which is discussed in Appendix A and illustrated in the following example where `f2` extends `f1`.

```
facet f1:: statebased
 hour: int;
 begin
  T0: 0<=hour<12;
end f1

facet f2:: statebased
 hour: int;
 minute: int;
 begin
  L0: hour=2 OR hour = 6;
  L1: minute=30;
end f2
```

The formal semantics must therefore take into account exactly when two facets should be considered as equivalent (point 1), and how the satisfaction relationship is affected by adding or removing constraints (point 2).

3.2.1 Formalisms satisfying these tenets

The first tenet of Section 3.2 can be satisfied by using *institutions*, introduced in Section 2.3.3. As well as providing increased generality, institutions allow us to formalise the notion that truth should be invariant under change of notation. That is, using institutions we can formally consider two syntactically different specifications to be equal, or to describe the same system. This is what is required by the first tenet of Section 3.2. In the language of institutions, we say that two sets of sentences $\{S\}_i$ and $\{S'\}_i$ are satisfied by the same models if there is an invertible signature morphism α from the vocabulary of $\{S\}_i$ to the vocabulary of $\{S'\}_i$, and if the elements of either set $\{S\}_i$ and $\{S'\}_i$ are implied, under this morphism and using the logic of the institution, by the elements of the other.

The second tenet of Section 3.2 can be satisfied by the use of theories to model specifications. That is, if there is a theory morphism $\gamma : Th_A \rightarrow Th_B$, then the set of models which satisfy B is contained within the set of models which satisfiy A. This means that a component (model) which satisfies one specification will satisfy all specifications of which this is a refinement. Theories are defined over institutions, which themselves satisfy the first tenet. That is, this proposal of using theories and institutions satisfies both our requirements for a semantic basis.

To provide the semantics of Rosetta, we propose associating each specification with a theory defined over the algebraic institution, and using theory morphisms to explicitly model the concept of facet extension. With this in mind, in the following section we describe how to associate a Rosetta facet with a theory, and express the relationships between Rosetta facets by using theory morphisms.

This can then be extended to other languages, allowing system analysis to take place within a framework which supports the syntax-independent representation of components and their relationships.

3.3 Rosetta, Institutions and Theories

Section 3.2 lists two central requirements for any formal semantic basis for the specification languages we examine, and for Rosetta in particular. In this section, we show how the category of theories defined over the algebraic institution satisfies these requirements. Firstly, we show how to obtain a signature and closed presentation (closed set of axioms) from each Rosetta facet. These will be used to generate the categories and morphisms which will make up the *Rosetta institution*. Furthermore, these signatures and closed presentations obtained from facets will also be used to define the category of theories over the Rosetta institution. We will then demonstrate how these theories help us formalise Rosetta interactions, as well as more general ways of combining components.

3.3.1 Facet Signatures and Presentations

Definition 3.1. *The* signature *associated with a Rosetta facet* `f1` *is a tuple* $\Sigma_{f1} = \langle S_{f1}, \Delta_{f1} \rangle$ *where*

- S_{f1} *consists of the names of the datatypes visible to* `f1`
- Δ_{f1} *consists of a set of operators denoting the variables, functions and constants visible to* `f1`.

S_{f1} includes the datatypes which are visible to `f1` via facet extension, as well as those datatypes declared within `f1`. Likewise, Δ_{f1} consists of all functions, variables and constants visible to `f1`. Specifically, each Rosetta variable x of type T within `f1` is denoted by a function $x \in \Delta_{f1}$ where x : `f1-state` $\rightarrow T$. Rosetta functions are also represented in Δ_{f1} as functions of the appropriate rank and arity. Any signature associated by Definition 3.1 with a Rosetta facet will then be an algebraic signature.

3.3.2 Rosetta Institutions

Building on this generation of an algebraic signature, we define the following institution for the purpose of analysing Rosetta facets.

Definition 3.2. *The Rosetta institution* $\mathcal{R}$ *contains the following elements:*

- *The category* ***RSig****, which is the category of all Rosetta signatures, generated as described in section 3.3.1. Morphisms within this category are algebraic signature morphisms.*
- *The functor Alg :* $\boldsymbol{RSig} \rightarrow \boldsymbol{Cat}^{op}$ *associates each signature* Σ *with the category of* Σ*-algebras. This is a standard functor used when discussing algebras.*

- *The functor RSen* : ***RSig*** $\rightarrow$ ***Set*** *associates each signature* Σ *with the set of* Σ*-sentences which may be legally expressed in Rosetta. These are the Rosetta axioms discussed in Appendix A.*
- *The satisfaction relation for this insitution is the equational logic satisfaction relation* $\models$*, relating* Σ*-models and* Σ*-sentences.*

As this institution, $\mathcal{R}$, is so closely related to the equational logic institution, we refer the readers to [31] for further discussion. We will use the category of theories defined over this institution to provide the formal semantics of Rosetta. However, for this to be useful we must first identify a way to associate a Rosetta facet with such a theory, and establish what these theory morphisms will mean in the context of Rosetta facets.

3.3.3 Rosetta Theories

Just as Definition 3.1 associated a Rosetta facet with an algebraic signature, we can associate a facet with a *presentation*, or family of constraints.

Definition 3.3. *The* presentation E^*_{f1} *associated with* `f1` *consists of the equational logic closure of all Rosetta axioms visible within* `f1`*.*

The presentation associated with `f1` includes the axioms which are within `f1` itself, as well as those axioms it inherits from domains or other facets. It is important to note that the presentation E^*_{f1} does *not* include axioms from any facet `f2` constraining the values of variables shared between `f1` and `f2`. For example, given the code of Example 1 in Appendix A, the presentation for `f1` would not include the axiom $L0$ from `f2`. Presentations must be equationally closed, as indicated by the Kleene-closure *-notation. That is, if it is possible for an equation $t_1 = t_2$ to be derived by equational logic from the axioms visible to `f1`, then this equation $t_1 = t_2$ should be included within E^*_{f1}. As Rosetta develops in complexity, we anticipate that equational logic will no longer suffice, and instead will be replaced with a higher-order logic, the change being reflected in the Rosetta `logic` domain.

Definitions 3.1 and 3.3 describe how each Rosetta facet `f1` can be denoted by a presentation $\langle \Sigma_{f1}, \mathrm{E}^*_{f1} \rangle$, where Σ_{f1} is the signature of this facet, and E^*_{f1} is closed under equational logic. Given the institution $\mathcal{R}$ of Definition 3.2, we obtain the following result.

Lemma 3.4. *Each Rosetta facet* `f1` *can be denoted by a theory* $\langle \Sigma_{f1}, \mathrm{E}^*_{f1} \rangle$ *over the institution* $\mathcal{R}$*.*

By using a semantic basis which associates a facet with a theory, we define facets to be unique up to axiom abstraction. By contrast, the names of variables, functions and constants can uniquely determine a facet, using these semantics. This is because the signature Σ of each theory is defined uniquely, rather than up to signature abstraction (although theory morphisms can be used to implement signature abstraction, if this is required).

The reason for allowing a signature to uniquely determine a facet is that signature abstraction generalises away from the issues involved in renaming variables. While these issues may be of little importance in pure theory, in practice they can introduce very complex problems. By avoiding the use of signature abstraction, we ensure that the theoretical framework will recognise this additional practical complexity. We can summarise the effects of these choices as follows.

Remark 3.5. *Under the proposed semantic basis, if two facets generate identical signatures and the axioms of either facet can be deduced by equational reasoning from the axioms of the other, then these two facets will be associated with the same theory. If two facets generate different signatures then, irrespective of their axioms, they will be associated with different theories.*

We can use the category of theories defined over the institution $\mathcal{R}$ to formalise any relationships which hold between two facets. In particular, we can describe the relationship between a facet and its extensions, and the relationship between equivalent facets. These relationships comprise the foundations of the two tenets of Section 3.2.

3.3.4 Formally Relating Rosetta Facets

By associating each Rosetta facet with a theory, we have laid the groundwork for a semantics based upon the category of theories. The two central tenets of Section 3.2 describe some informal conditions for equivalence of facets, and some implications of facet extension. Within this section we will show how these tenets can be satisfied using theories as a semantic basis. Rosetta facets and interactions will be used to demonstrate this satisfaction, although the issues we discuss must be considered when developing a semantic basis for any language.

Firstly, we focus on a subcategory of the category of theories defined over $\mathcal{R}$. This, **RTh**, will be sufficient for examining Rosetta facets.

Definition 3.6. *The category **RTh** is comprised of*

- *objects which are those theories $\langle \Sigma_{f1}, \mathrm{E}^*_{f1} \rangle$ identified with a Rosetta facet, as introduced in Lemma 3.4*
- *morphisms which are those signature-injective (monic) morphisms within the category of theories defined over $\mathcal{R}$.*

Signature-injective morphisms were discussed in Section 2.3.3. This definition implies that if $\alpha : Th_a \rightarrow Th_b$ is a morphism within **RTh**, then the signature of Th_b must contain the signature of Th_a, up to any renaming which preserves the distinction between different elements in the signature of Th_a. We now show how the properties of facet equivalence and facet extension (as described in the tenets of Section 3.2) are interpreted within **RTh**.

Equivalence of Facets (Tenet 1)

The definition of equivalence in Rosetta, introduced in Section 3.2, is stricter than the general usage. Informally, two facets are equivalent if they have the same number and type of variables, and after applying some renaming function the closure of the set of axioms of each facet implies the closure of the set of axioms of the other. More specifically, two facets must contain exactly the same number and type of data elements (such as constants, functions and variables) in order to be deemed equivalent Rosetta facets. Furthermore, the constraints of each facet must imply the constraints of the other under equational logic. The following definition encapsulates this informal requirement.

Definition 3.7. *Two facets f1, f2 denoted by theories* $\langle \Sigma, E^* \rangle$, $\langle \Sigma', E'^* \rangle$ *are* equivalent *if and only if there is a theory morphism* $\alpha : \Sigma \to \Sigma'$ *where*

- α *is invertible when considered as a signature morphism*
- $\alpha(E)^* = E'^*$

From this, we derive the following lemma.

Lemma 3.8. *Two Rosetta facets are equivalent if and only if they generate isomorphic theories in* ***RTh****.*

Extension of Facets (Tenet 2)

Rosetta extensions consist of addition of data elements and constraints. In other words, if facet `f1` extends facet `f2`, then `f1` will contain all the constraints of `f2`, and also all the data elements of `f2`. This means Rosetta facet extensions are represented by those theory morphisms which are signature-injective. When taken in the context of the category **RTh**, we obtain the following lemma.

Lemma 3.9. *Rosetta facet extension is modelled by theory morphisms in* ***RTh****.*

The second tenet of Section 3.2 describes how the satisfaction relationship should interact with the concept of facet extension. Briefly, a system satisfying a facet `f1` should satisfy `f2`, if `f1` extends `f2`. If morphisms in **RTh** are to model Rosetta facet extensions, then we must prove that they satisfy this requirement. This can be done by referring back to the underlying Rosetta institution, $\mathcal{R}$.

In order to interpret the satisfaction relationship within the category of theories, we first note that for a given theory $\langle \Sigma, E^* \rangle$, the family E'^* is a family of sentences $\{e\}_i$. The satisfaction relation $\models$ relates an algebra m to the set of sentences $\{e\}_i$ if and only if m satisfies every $e_i \in E^*$. It is clear that the theory morphisms in **RTh** denote adding axioms or signature elements to a theory $\langle \Sigma, E^*, \rangle$ to obtain $\langle \Sigma', E^{*\prime} \rangle$. The following lemma formally relates the action of such a morphism to the set of models of each theory.

Lemma 3.10. *Let α be a morphism in* ***RTh****, such that $\alpha : \langle \Sigma, E^* \rangle \rightarrow \langle \Sigma', E^{*\prime} \rangle$. Let A be the set of models (algebras) of $\langle \Sigma, E^* \rangle$. This can be defined as the following set A:*

$$A = \{x \text{ such that } x \text{ is an object of } \mathrm{Alg}(\Sigma) \text{ and } x \models e, \text{ for all sentences } e \in E^*\}$$

Let B be the set of models of $\langle \Sigma', E^{*\prime} \rangle$. This can be defined as the following set B:

$$B = \{x \text{ such that } x \text{ is an object of } \mathrm{Alg}(\Sigma') \text{ and } x \models e', \text{ for all sentences } e \in E^{*\prime}\}$$

That is, for any $a \in A$ and $b \in B$, $a \models \langle \Sigma, E^* \rangle$ and $b \models \langle \Sigma', E^{*\prime} \rangle$.
Then $\mathrm{Alg}(\alpha)(b) \in A$, for any model $b \in B$, by definition of the satisfaction relationship $\models$.

That is, the set of models which satisfy the facet represented by $\langle \Sigma', E^{*\prime} \rangle$ is a subset of the set of models satisfying the facet represented by $\langle \Sigma, E^* \rangle$. This means that the morphisms α in **RTh** satisfy the informal requirements of the second tenet in Section 3.2. An example of a specification which is satisfied by multiple models is a specification which does not uniquely constrain some variables. Any model which assigns these variables values which do not violate any of the constraints is said to satisfy the specification. Thus, although each model assigns only one value to each of these variables, all possible legal values are represented in the set of models which satisfy the specification.

We have now constructed a semantic basis which meets the two fundamental requirements introduced in Section 3.2. This semantic basis is the category **RTh**, which is a subcategory of the category of theories defined over the Rosetta institution $\mathcal{R}$. Equivalent facets are represented by isomorphic theories within **RTh**, while facet extension is modelled by theory morphisms in **RTh**. This is formally supported by the construction of $\mathcal{R}$, as we see in Lemma 3.10. In the following section, we show how **RTh** can also be used to lend formality to a discussion of the relationships between components.

3.4 Relationships Between Components

The ability to express the relationships existing between components is fundamental to system analysis. We have already seen in Section 3.3.4 how one such relationship — facet extension — can be modelled by morphisms in **RTh**. In this section we demonstrate, again using Rosetta facets, how components may be related in other ways. In this process we will provide a formal representation of the types of Rosetta facet interaction discussed in Appendix A. In addition, we will show how designers can create their own components by combining existing specifications.

Any type of interaction between components becomes more complex when information sharing is necessary, because of the requirement that components interpret shared information consistently. As discussed in Section 2.2.1, this can be achieved by using a common, system-wide representation for shared data. Such a representation is known as a data universe. One consequence of using a data universe is that when combining two components `f1` and `f2`, there should be only one copy of the data universe in the combined specification. This, in turn, affects the constructions we use in the category of theories to model this combined specification. We will now formally describe the Rosetta facet interactions of Appendix A, using the category of theories as a semantic basis. In Section 3.5 we also provide a more general treatment of component interactions.

3.4.1 Facet Implication

Facet implication is one of the simplest pre-defined Rosetta interactions. A Rosetta facet `f1` is said to *imply* another facet `f2` if any axiom which `f1` includes is also included by `f2`. This means that any implementation satisfying `f2` will also satisfy `f1`. This can be expressed very simply in terms of **RTh**.

Definition 3.11. *Let `f1` and `f2` be denoted respectively by theories* $\langle \Sigma_1, \mathrm{E}^*_1 \rangle$ *and* $\langle \Sigma_2, \mathrm{E}^*_2 \rangle$. *Then `f1`* implies *`f2` if and only if there exists a morphism* $\alpha : \langle \Sigma_1, \mathrm{E}^*_1 \rangle \rightarrow \langle \Sigma_2, \mathrm{E}^*_2 \rangle$ *within **RTh**.*

3.4.2 Facet sum

Facet sum is one of the most common ways of combining Rosetta facets together. In its most basic form, facet sum describes the creation of a system wherein multiple components exist but do not communicate with each other beyond sharing common declarations from the data universe. Facet sum involves generating a new facet, `f1+...+fn`, which represents the sum of the original facets `f1, ..., fn` in the interaction. `f1+...+fn` is sometimes referred to as the *representational component* of the facet sum interaction, a concept to which we return in Section 5.5.1.

Given facets `f1,...,fn` we will define the representational component `f1+...+fn` of their facet sum by means of a colimit diagram D in the category of theories. This colimit diagram will contain the theories representing facets `f1,...,fn`, as well as a specific subset of their extension hierarchies. This specific subset will be the data universe shared by these components. This corresponds, in Rosetta, to the union of the domains extended by all of these facets.

Definition 3.12. *The colimit diagram D representing the sum of components $f_1, ..., f_n$ is the diagram in the category of theories consisting of these summands and their data universe.*

From this definition, we can obtain the object in the category of theories which corresponds to the sum of components `f1,...,fn`:

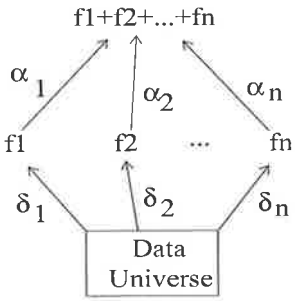

Figure 3.1: Facet sum represents an interaction in which the components share nothing but the data universe

Definition 3.13. *The* facet sum $f_1 + ... + f_n$ *is the facet denoted by the colimit object of D in the category of theories. We use the category of theories (of which* ***RTh*** *is a subcategory) to express this sum, as* ***RTh*** *itself may not necessarily have sums.*

This colimit diagram is shown in Figure 3.1.

Since the colimit object is defined up to isomorphism we see that the facet sum is determined uniquely only up to signature abstraction and axiom abstraction. Since all isomorphic theories over a given institution are associated with the same set of models, it does not matter, from a specification perspective, that we don't uniquely identify the facet representing `f1+...+fn`.

3.4.3 Examples of Facet Sum

Within this section, we provide an example of facet sum, showing exactly how the theory $\langle \Sigma, E^* \rangle$ of the representational component is generated. The summands here are two Rosetta facets `f1` and `f2` which share a common data universe. This data universe will be implemented as a Rosetta domain, `bool`, extended by both facets. For simplicity, we will choose one colimit object out of the many possible isomorphic objects.

We let the theory $\langle \Sigma_{bool}, \mathrm{E}^*_{bool} \rangle$, represent domain `bool`, while theories $\langle \Sigma_1, \mathrm{E}^*_1 \rangle$ and $\langle \Sigma_2, \mathrm{E}^*_2 \rangle$ represent facets `f1` and `f2` respectively. We can define theory morphisms which describe the extension of domain `bool` by facets `f1` and `f2`. These will be the theory morphisms

$$\delta_1 : \langle \Sigma_{bool}, \mathrm{E}^*_{bool} \rangle \rightarrow \langle \Sigma_1, \mathrm{E}^*_1 \rangle \text{ and}$$
$$\delta_2 : \langle \Sigma_{bool}, \mathrm{E}^*_{bool} \rangle \rightarrow \langle \Sigma_2, \mathrm{E}_2 \rangle$$

To identify the theory corresponding to facet sum `f1+f2` we use Definition 3.12, where the colimit diagram D consists of theories $\langle \Sigma_{bool}, \mathrm{E}^*_{bool} \rangle$, $\langle \Sigma_1, \mathrm{E}^*_1 \rangle$ and $\langle \Sigma_2, \mathrm{E}^*_2 \rangle$, and theory morphisms δ_1 and δ_2. This is shown in Figure 3.2.

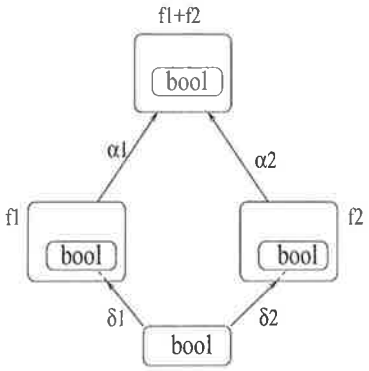

Figure 3.2: Colimits of theories which share elements

$\langle \Sigma_{f1+f2}, \mathrm{E}^*_{f1+f2} \rangle$ is the colimit object in this diagram. We note that the theory morphisms α_1 and α_2 do not describe any overlap with each other except for the data universe. That is, they do not map any data elements $f1.x$ and $f2.y$ which do *not* originate in the shared data universe to the same data element in `f1+f2`. We discuss the phenomenon of interference in different types of component interactions in Section 3.5.

The facets below illustrate this situation. `f1` and `f2` both extend the domain `bool` and therefore contain all its declarations and constraints.

```
facet bool
  bool: type;
  false, true: bool;
  +: bool * bool -> bool;
  begin
   T0: true+false = false;
end bool
```

```
facet f1:: bool
  char: type;
  const 'a': char;
  ischar: char -> bool;
  begin
   A1: ischar('a') = true;
end f1
```

```
facet f2:: bool
  int: type
  const 0: int;
  isint: int -> bool
  begin
   B1: isint(0) = true;
end f2
```

The theory morphisms δ_1 and δ_2 include `bool` in `f1` and `f2` respectively. We now generate the Rosetta facet corresponding to the theory $\langle \Sigma_{f1+f2}, \mathrm{E}^*_{f1+f2} \rangle$,

using Definitions 3.1 and 3.3. This will be the facet containing the following declarations and constraints (up to axiom abstraction).

```
facet f1+f2
  int:type; char:type;
  const 'a': char; const 0: int;
  ischar: char -> bool; isint: int -> bool
  bool: type;
  false, true: bool;
  +: bool * bool -> bool;
  begin
   T0: true+false = false;
   A1: ischar('a') = true;
   B1: isint(0) = true;
end f1+f2
```

As we can see, facet `f1+f2` contains only one copy of the declarations in domain `bool`. Because `f1+f2` extends the facets `f1`, `f2` and `bool`, we may instead write the above declaration as

```
facet f1+f2:: bool, f1, f2
end f1+f2
```

While this form of writing does illustrate the point made in Appendix A, that Rosetta facet extension is transitive, it is less helpful in illustrating the meaning of facet sum.

3.5 Relating Components Using Interference

In this section, we discuss how to use the category of theories to model general component interaction. We will no longer restrict ourselves to a Rosetta-specific context, but instead apply these findings to all languages which are amenable to using theories as a semantic basis. In general, when we create a new component from existing communicating components, we must consider how this communication affects elements of the new component. This is known as *interference*, and will be formally represented using diagrams in the category of theories. The construction of these diagrams depends upon how the components interact. For example, the designer may want a single variable of the representational component to be bound by constraints from several of the interacting components. This is equivalent to requiring that the theory morphisms α_1 and α_2 of Figure 3.2 act on variables x in `f1` and y in `f2` such that $\alpha_1(x) = \alpha_2(y)$. In this case, the variable $\alpha_1(x)$ in the representational component `f1+f2` would be bound by all constraints that `f1` applied to x *and* all constraints which `f2` applied to y. Notwithstanding the number of different interactions which can occur between multiple components, we still wish to use colimits in the category of theories to describe each interaction. The following section describes a formal technique for constructing diagrams which allow this.

Allowing interference when combining components means that we can create systems with a wider variety of properties. For example, a designer might specify a system as a combination of components, each of which contains some common basic element such as a switch. Moreover, each of these component might constrain its switch differently. The system resulting from a combination of these components might contain one highly-constrained switch, or alternatively might contain many differently constrained switches. That is, we can create a component which has the maximum number of variables all minimally constrained, or the minimum number of variables all maximally constrained — or some other combination upon this spectrum. In the following section, we discuss how each of these different systems can be formally described as a combination of the original components.

3.5.1 A Formal Treatment of Interference of Variables

When formally representing component combinations, we will use colimit diagrams in the category of theories to define the resultant system in each case. This implies that any interference should be represented in these diagrams. We will do this by creating a 'virtual' theory containing those data elements from each summand which interfere. The summands in question will extend this virtual theory, the extension being shown in the relevant colimit diagram. The colimit object of this diagram will then, by commutativity considerations, demonstrate the desired interference.

In general, when summing components `f1,...,fn` where we want to establish interference of variables x and y from summands `fi` and `fj` respectively, we model this by defining a smaller, 'virtual' theory for each pair (`fi`, `fj`) of interacting components. This theory declares one variable xy for each pair (x, y) — from `fi` and `fj` respectively — affected by interference, but includes no constraints upon the variable xy.

Construct 3.14. *Assume the variables $x_1, ...x_m$ from `fi` and $y_1, ...y_m$ from `fj` are intended to undergo interference pairwise (x_k, y_k) when combining `fi` and `fj` in a single system wherein the data universe is represented by a theory $\langle \Sigma_D, \Upsilon_D \rangle$. We then define a* virtual theory $Th_{xy_1,...,xy_m}$ *as*

$$Th_{xy_1,...,xy_m} = \langle (\Sigma_D \bigcup (xy_1, ..., xy_m)), \Upsilon_D \rangle$$

where a single variable xy_k is declared in $Th_{xy_1,...,xy_m}$ for each pair of interfering variables from `fi` and `fj` respectively.

The virtual theory contains the data universe $\langle \Sigma_D, \Upsilon_D \rangle$ (for example, in Section 3.4.3, this theory would represent the Rosetta domain `bool`), and a declaration of a single variable xy_k for each pair (x_k, y_k) affected by interference. The obvious theory morphisms $\delta_1 : Th_{xy_1,...,xy_m} \rightarrow Th_{fi}$ and $\delta_2 : Th_{xy_1,...,xy_m} \rightarrow Th_{fj}$ respectively map xy_k to x_k in `fi`, and to y_k in `fj`. The theory $Th_{xy_1,...xy_m}$ is then included, with theory morphisms δ_1 and δ_2, into a diagram D in the

category of theories. A new virtual theory is defined for each pair of components demonstrating interference.

Construct 3.15. *The diagram D representing the interaction of components* `f1, . . . , fn` *comprises the theories $Th_{f1}, ...Th_{fn}$ representing these components, the data universe theory $\langle \Sigma_D, \Upsilon_D \rangle$ and any virtual theories $Th_{xy_1,...,xy_n}$ representing interference.*

The component resulting from this interaction is then the colimit object of D. This is known in Rosetta as *relative facet sum*, or more generally the *representational component* of `f1, ..., fn`.

Definition 3.16. *The* representational component *of components* `f1, . . . , fn`*, where* `fi` *and* `fj` *interfere on data elements $(x_1, y_1), ...(x_n, y_n)$ as in Construct 3.14, is the colimit object of the diagram D constructed according to Construct 3.15. It is written as the theory Th_{Rep}, where*

$$Th_{Rep} = Th_{f1} \times ... \times Th_{fi} \times_{xy_1,...xy_m} \times Th_{fj} \times ... \times Th_{fn}.$$

To identify this representational component in the category of theories, the designer must provide information about the desired degree of overlap.

An Example of Interference of Variables

The following Rosetta code provides an example of this scenario. Here, both components `f1` and `f2` include a declaration of a switch, although they constrain it differently.

Switch Example

```
facet f1:: state-based
  private switch: int;
  begin
   T0: switch' > 5;
   ...
end f1

facet f2:: state-based
  private switch: int;
  begin
   L0: switch' < 10;
   ...
end f2
```

We can then specify a facet interaction in which these are combined to produce a facet containing a single, highly-constrained switch. When modelling this in the category of theories, we define a 'virtual' theory Th_{switch}, which includes the data universe and a single *switch* variable. Th_{switch} places no constraints on this variable. That is, the facet corresponding to Th_{switch} has the form

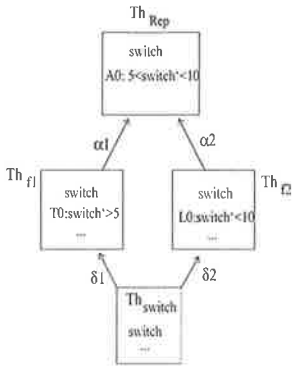

Figure 3.3: The representational component Th_{Rep}, known in Rosetta as the relative facet sum

```
facet switch:: state-based;
  private switch: int;
  begin
end switch
```

The representational component Th_{Rep} is the colimit of Th_{switch}, Th_{f1} and Th_{f2}, as shown in Figure 3.3. The Rosetta code for such a facet `Rep` is as follows:

```
facet Rep:: state-based
  private switch: int;
  begin
   A0: 5 < switch' < 10;
   ...
end f3
```

3.5.2 The advantages of formal interactions

The advantage of using such a formal technique based upon the category of theories is that we are now able to identify the similarities and relationships between two interactions which might otherwise appear unrelated. This applies in any language which can make use of the category of theories as a semantic basis. This technique also provides a precise and categorical description of a customized interaction in any of these languages, provided the designer supplies information about the areas of overlap. This is in contrast to an *ad hoc* approach whereby a designer must try and express the relationship between components informally.

One of the less obvious disadvantages to an informal expression of relationships is confusion over whether certain information originated in the data universe, or in one of the components. That is, given a representational component `f1` $\times_{xy}$ `f2`, without examining the corresponding diagram in the category of theories, we cannot tell which information in `f1` $\times_{xy}$ `f2` is part of the shared data universe of `f1` and `f2`. Being able to identify this information is important because, as noted in Appendix A, components may not alter the data universe by adding constraints to it. If `f1` $\times_{xy}$ `f2` is later to be used as a component within a larger system, we must ensure the system does not attempt to constrain any information originating in the data universe of `f1` $\times_{xy}$ `f2`. This can most easily be done by examining the diagram in the category of theories which represents this relative facet sum.

3.5.3 A Customized Data Universe for Specification Languages

Throughout this thesis, we will be making a number of assumptions about the systems being specified and analysed. For example, all the systems we examine in detail will be state-based. Another assumption, which affects the composition of the data universe, relates to the details of specifying a datatype. When writing specifications, it is common to specify datatypes constructively. These include the generator inductive definitions of ABEL [18] and Rosetta, as well as Rosetta's enumerated types. Such datatypes include enumerated types:

```
Months = [Jan, Feb, Mar, ...];
```

and any datatype specified by means of a generator function:

```
datatype natural;
zero: -> natural;
succ: natural -> natural;
```

For these to be implemented correctly, we require that elements of these datatypes will not be confused in any specification. For example, a designer specifying a system works on the principle that each application of the `succ` function will produce a new element of the naturals. As we have discussed in Section 2.3.1, this can be achieved by including meta-constraints which require certain parts of the specification to be modelled as an initial algebra. In this work, these parts of the specification will only be those which define datatypes, and usually only those datatypes which are part of the data universe.

A crucial point here is that the *entire* data universe need not be modelled as an initial algebra. The simplest example of such a situation occurs when declaring constants within the data universe. Namely, it is possible to declare a constant within a Rosetta domain (ie. within the data universe) and deliberately avoid including sufficient axioms within this domain to uniquely constrain it.

A meta-constraint requiring that the entire data universe be modelled as an initial algebra would remove any possibility of this low-level non-determinism applying to constants within the data universe. In the following chapter, we demonstrate how to enforce these meta-constraints categorically, leading to a more homogeneous formal framework.

3.5.4 Theories and Institutions

Within this chapter, we have demonstrated the utility of a semantic basis predicated upon the category of theories, for the class of languages specified by CoFI. Although we have used Rosetta to provide examples, the definitions and results of this chapter hold for any specification language of this class. The semantics we have proposed uses institutions to capture the idea that truth should be invariant under change of notation. That is, two specifications which differ only in variable names, or using equivalent but syntactically different axioms, are satisfied by the same set of components. By associating each specification with a theory, we can use theory morphisms to explicitly model the structural relationships between components. However, this does not provide any information about dynamic interaction between the components, such as behaviours that can synchronise. That is, while the theory morphisms denote which variables are shared in a system, they do not tell us what this implies about the behaviours of the components which share these variables.

In the following chapter, we provide a category theoretic treatment of system state. This permits us to explicitly examine behaviours of individual components. The relationships between components will become more complex, and we will express them in terms of behaviour rather than structure. As a result, we will be able to deduce more about the possible inconsistencies within a system. This proposed category theoretic analysis does not supersede the use of theories and institutions presented in this chapter. Rather, the framework proposed in the next chapter is intended to be used alongside the category **RTh**, as a complementary technique for formal analysis. Within Chapter 5 we shall show how the two frameworks can be used together, to provide both dynamic and structural analysis.

Chapter 4

The Categorical Consistency Framework

In the previous chapter we have shown how the category of theories can be used as a semantic basis for Rosetta and similar languages. This semantic basis provides a formal and cohesive technique for describing the relationships between the components of a system. However, the analysis provided so far using the frameworks of institutions and theories has been structural rather than dynamic. That is, we have analysed the *structure* of systems formed from multiple components, but we have not examined how the behaviour of one of these components affects the behaviours of the others. The analysis of behaviour is arguably more important than the analysis of structure when it comes to inconsistency management. For example, one common behavioural analysis problem is to identify whether or not a particular component can behave consistently in a proposed environment. We also wish to know exactly what types of inconsistency can occur, and how these types are characterised.

The framework we use for this dynamic analysis should not only be detailed enough to examine inconsistencies, but also flexible enough to be used for many different specifications. For example, different specifications use different notions of state, and any formal framework should be able to describe all of these. Similarly, when identifying superfluous components in a system, we perform a different type of analysis to that performed when we attempt to deduce system characteristics from a subsystem. The formal framework we use should be able to explicitly express these different analysis operations. Ideally, each new behaviour or property that we wish to analyse should not necessitate the generation of a new theoretical framework. This would be time-consuming, and each iteration would inevitably lack the facility to identify connections and similarities between the different behaviours and analysis techniques.

Similarly, the framework we use should be able to identify inconsistencies and their causes, as well as suggest some metrics for determining their effects. To begin with, it should explicitly describe the conditions required for each individual state of a system to be consistent. This means that, given a state, the framework can be used to identify whether or not this state, considered in isolation, violates any of the system constraints. However, inconsistencies can occur in more than a single state. For example, a behaviour might be inconsistent even though each individual state fulfills consistency requirements. The framework we choose should also be able to identify these types of inconsistency and suggest resolution mechanisms.

In this chapter we propose the *Categorical Consistency Framework* (CCF) for the dynamic analysis of systems. This is a framework based on the notions of category theory and EA sketches. It permits us to analyse system behaviours, while providing the detail and flexibility discussed above. Section 4.1 provides an overview of CCF, along with a guide as to the section numbers where more detail on each process can be found. We then discuss the variations between different systems' definitions of state in Section 4.2. This section also provides

the definitions and assumptions about state that we will use throughout this chapter. Section 4.3 and 4.4 then describe precisely how the fundamental categories of CCF are constructed, given these definitions. Section 4.5 introduces the ways in which an individual state might be inconsistent. These are presented in terms of CCF, showing how this framework can be used to identify such inconsistencies. Section 4.6 then demonstrates how to use CCF to produce a category of consistent states, in which morphisms represent the behaviours of the system. Finally, Section 4.7 provides a taxonomy of the different types of inconsistencies (both behavioural and state-specific) and how they can be detected using CCF.

While the discussion so far has been tailored for Rosetta, the concepts are applicable to any modular specification language. This will be the case throughout this work, with Rosetta used only to provide practical applications of the general concepts we introduce.

4.1 Overview of CCF

The Categorical Consistency Framework (CCF) consists structurally of a number of categories and functors, each representing a different aspect of the system in question. The categories are all generated by extending the notion of an EA sketch. As mentioned in Section 2.4.3, these sketches have historically been used to represent the relationships between entities or data. We begin the construction of CCF by first defining a type of sketch we will term a *CCF sketch*. A CCF sketch is similar to an EA sketch, and is defined as follows:

Definition 4.1. *A CCF sketch $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ is a finite limit, countable sum sketch [9] consisting of:*

- *A directed graph G*
- *A family $\mathcal{D}$ of some of the pairs of paths in G with a common source and target*
- *A set $\mathcal{L}$ of finite cones [55] within G, including the cone with empty base*
- *A set $\mathcal{C}$ of finite cocones [55] within G*

This CCF sketch presents the *system category* $\mathbf{C}$, under conditions described in Section 2.4.3. The category $\mathbf{C}$ encapsulates all the information provided in a system specification. Specifically, datatypes are implemented as objects of $\mathbf{C}$, while variables are implemented as morphisms. The system's constraints are reflected in $\mathbf{C}$ as equalities of the appropriate morphisms. In Section 4.3 we show how to construct a CCF sketch representing a system. We also discuss the difference between EA sketches and CCF sketches in more detail, showing why the extension to EA sketches is required.

In Section 4.4 we consider what is necessary to represent a single state of a system categorically. We show how to construct a second CCF sketch which represents such a state. For example, in each state every variable within a system must have a single value, a property reflected in the construction of this second CCF sketch. This CCF sketch is then used to generate the *abstract state* category $\mathbf{C}_V$. A state of the system is then a functor R from this abstract state category $\mathbf{C}_V$ to **Set**. This functor associates each variable represented in $\mathbf{C}_V$ with a single value of the correct datatype in **Set**. That is, this functor is a mapping from variables to values, in keeping with the traditional definition of state.

As we discuss in Section 4.5, without any restrictions on the action of this functor R there is no way to ensure that it will represent a consistent state. If this is to be the case, then firstly the constraints of the system embodied in the system category $\mathbf{C}$ must be satisfied. We achieve this by requiring that R factor through $\mathbf{C}$, so that the equalities in $\mathbf{C}$ (which represent constraints) are applied to R. Secondly, in a consistent state every variable has only one value, no matter how many components can access it. To enforce this, we define an *interaction consistency* category $\bar{\mathbf{C}}_V$, which ensures that two components agree on a single value for any shared variables in any given state. Requiring that R factor through $\bar{\mathbf{C}}_V$ ensures that the constraints of $\bar{\mathbf{C}}_V$ are satisfied in the state represented by R. Thus, if R is to represent a consistent state, it must factor through both $\mathbf{C}$ and $\bar{\mathbf{C}}_V$. In Section 4.6 we define the category of states R of a system. Objects in this category are consistent state functors R, while morphisms represent behaviours of the system. This category of states, and the categories $\mathbf{C}$, $\mathbf{C}_V$ and $\bar{\mathbf{C}}_V$ described above, form the structure of CCF.

Finally, in Section 4.7 we show how CCF can help us identify and resolve different types of conflict. In the process we create a taxonomy of inconsistencies. This includes inconsistencies which occur only in a single state, as well as inconsistencies which become obvious only when behaviours are examined. In addition, this taxonomy includes a discussion of different means of resolving each type of inconsistency. We discuss the role of user input, most particularly the identification of 'essential' behaviours or states, which must not be affected by inconsistency if this can be avoided. We wil return to this topic throughout the thesis.

4.2 Defining States and Behaviour

The question of exactly what distinguishes different states of a system has been discussed extensively in the literature [2, 45]. The simplest definition, which we adopt in the remainder of this chapter, is that a state is distinguished only by the (visible) values of variables. In Chapter 5 we present some variations upon this basic definition. These include systems which use abstraction mechanisms

and therefore define equality of states as equivalence, as well as systems which present different means of distinguishing behaviours.

Another fundamental property of a system is the way in which the variables are constrained. For example, it is very common for the values in the 'next' state to be constrained relative to the values in the current state only. In this case, observing the values in the current state is sufficient to predict the next state, modulo any non-determinism in the system. However, though less common, it is possible to write specifications where the values of variables in a state are constrained relative to values in another, arbitrary, state. In Chapter 5 we present a Rosetta-specific system specification of this latter type.

In this chapter, however, we will restrict our attention to the simplest definition of state. We show how this can be modelled using CCF, and discuss how this framework permits us to characterise and distinguish behaviours.

4.2.1 A Basic State Definition

The systems we analyse in this chapter will all be Basic State systems, satisfying the following informal definition.

Definition 4.2. *A* Basic State *system specification is one which satisfies the following assumptions:*

- *A component's state is distinguished by the current values of its variables.*
- *The values in the next state are constrained relative to the values in the current state only.*
- *The value of variables in the initial state may be constrained relative to constants or functions applied to constants.*
- *Behaviours are distinguished by the states observed.*

Similarly, a *Basic state* is a state of such a system, and a *Basic behaviour* is a behaviour observable in this system. Under these assumptions, variables cannot be constrained relative to values in an arbitrary state (such as an axiom `x@s10 = y@s20`), nor do we permit such axioms as `x'' = x+1`, which relate the value of a variable after two state transitions to a current value. The constraints within a Basic State system will therefore appear in the form $t1 = t2$, where $t1$ and $t2$ are terms consisting of either

1. functions or constants
2. variables in the initial state
3. variables
4. function applications to any of the above, where multiple applications of the 'next' function are disallowed

All specification languages in the class we examine can be used to specify Basic State systems. However, we note that as shown in examples in Appendix A, Rosetta can express axioms which would define a system which does not conform to Definition 4.2. We will discuss these systems further in Chapter 5.

Definition 4.2 is intended only as an informal definition. In the following sections we will show how to formalise each point of this definition.

4.3 System Specification Using a CCF Sketch

The Categorical Consistency Framework associates a system specification with a number of categories. Perhaps the most fundamental of these is the *system category* **C**, which represents every declaration and axiom of the specification. In order to generate this category, we must first represent the system in a more suitable form than that provided by a specification language. The representation we choose is that of an CCF sketch, introduced in Definition 4.1. CCF sketches are themselves based on EA sketches. One reason for this choice is that the EA sketch formalism lends itself easily to modelling views of an entity, database or system — a capability not offered by all formalisms. A view of the system corresponds to the perspective of a particular component or subsystem. This means that EA sketches — and by extension, CCF sketches — provide a structure which can be used to identify subsystems and their behaviours, a topic to which we return in section 5.3.

4.3.1 The CCF Sketch Representing a System

We demonstrate here how to construct a CCF sketch $Sys = (G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ which represents a system specification. We assume the specification in question comprises a number of components `f1, ..., fn`. We will show how each element of this tuple is constructed, and how these elements together represent a system.

Definition 4.3. *The CCF sketch $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ representing a system specification comprised of components `f1, ..., fn` is a tuple consisting of:*

- *A directed graph G whose nodes represent the datatypes of the specification and whose edges represent the variables, functions and constants of the specification*
- *A family $\mathcal{D}$ of some of the pairs of paths in G with common source and target, representing the axioms of the specification*
- *A family $\mathcal{L}$ of some of the cones in G*
- *A family $\mathcal{C}$ of some of the cocones in G*

We discuss these elements in more detail below, making reference to the following code.

Example 1

```
facet f1 :: state-based
 public x: int;
 private y: int;
 begin
  T0:x@s0 = 0, y@s0 = 0;
  T1:y' = y+1;
end f1
```

```
facet f2 :: state-based
  begin
   L0: x' = x+1;
end f2
```

The Graph G Representing A Specification

Clarification 4.4. *In Definition 4.3, the graph G contains nodes which represent:*

- *The datatypes declared in the system, such as* `int`, `fi-state` *(for each component* `fi`*),* `char` *etc.*
- *Any product type of the above datatypes, eg.* `int` $\times$ `character`.
- *A special terminal node labelled* 1*, which will serve as a* `null` *datatype*

G contains edges which represent the functions, variables and constants which make up each component, but does not include the `current` *variable (of type* `state`*) which is part of all Rosetta specifications.*

The graph G constructed in this way represents an entire system statically, with variables being treated as functions of the `state` datatype. An edge from one node N_1 to another N_2 represents a function with domain N_1 and range N_2. Constants may be represented as functions with domain 1, while any variable of component `fi` is represented as a function from the node representing the state of `fi` (the `fi-state` node) to the node representing the appropriate datatype. We exclude from G the variable `ficurrent` of type `fi-state` for each Rosetta component `fi`. In Section 4.3.5 we will demonstrate how this variable is used, but for now we simply note its omission from G. In determining the nodes and edges in G, we recall the earlier discussion of the data universe in Section 3.4. Here, we determined there can be only one copy in the system of the data universe. This is achieved in CCF by defining only one node in G to represent a datatype in the data universe, regardless of how many components share this datatype.

Figure 4.1 shows part of the graph G representing the code of Example 1. Strictly speaking, this shows part of the category based on the graph, but these appear identical in diagrams. Figure 4.1 illustrates how shared variables are implemented under this framework. Specifically, given a variable x visible to two different components `f1` and `f2` but not part of the data universe, as in Example 1, x is represented in the graph G as a distinct variable $f1.x$ in `f1` and $f2.x$ in `f2`. Thus, each shared variable viewed by multiple components is

represented in G by multiple edges. This echoes the structural semantic basis provided in Chapter 3, wherein each component was represented by a separate theory, meaning a shared variable was declared in multiple theories.

In addition to the graph G, we also need a means of representing the constraints placed upon variables. This is achieved by the use of the family of pairs $\mathcal{D}$.

Equalities and the Family $\mathcal{D}$

Clarification 4.5. *In Definition 4.3, the family $\mathcal{D}$ consists of some of those pairs of paths in G with common source and target. These pairs represent those pairs of functions, variables and constants which are constrained to be equal by the axioms of the specification.*

The construction of these pairs of edges to represent functions and constraints is relatively straightforward. The pairs are generated directly from the axioms for each component. However, functions and constraints may be quite complex, such as `IF` or `WHILE` expressions. It has been demonstrated in [51], based on a formalism by Hoare [50], that it is possible to express these imperative language functions within a categorical framework. While it is beyond the scope of this work to incorporate a full discussion of the categorical equivalents of imperative language constructs, we refer the reader to [51] for further details.

Every pair of paths in $\mathcal{D}$ is obtained directly from the axioms for each component. $\mathcal{D}$ also includes the axioms constraining the data universe (the axioms within Rosetta domains, for example). For example, if the data universe includes a definition of the natural numbers defined via the *succ* and 0 functions, one axiom within $\mathcal{D}$ might be

$$succ(0) \neq 0$$

Constructing $\mathcal{D}$ in this way ensures that each component observes the same constraints upon the data universe, and hence that all components interpret the data universe in the same way. Because any pair in $\mathcal{D}$ must be associated with an axiom found in a component, there are no constraints represented in $\mathcal{D}$ which pertain to state synchronisation of multiple components. That is, there are no axioms represented in $\mathcal{D}$ which explicitly constrain what state one component `f1` must be in, relative to the state of another component `f2`. The omission of these axioms from this CCF sketch allows us to consider inconsistent systems and states within this framework.

Limits $\mathcal{L}$

Construct 4.6. *The family $\mathcal{L}$ consists of those cones in G in which the datatype represented by the apex object is the product of the datatypes and functions represented by the base objects.*

It is possible for the cones in $\mathcal{L}$ to vary from one specification to the next. This is because $\mathcal{L}$ represents limit cones, and limit cones can also be used for other purposes, such as ensuring that a function is monic. However, in this thesis, our use of limit cones will be — as reflected in this Construct — purely to identify those datatypes which are defined as the product of others.

Colimits $\mathcal{C}$

Construct 4.7. *The family $\mathcal{C}$ consists of cocones representing those datatypes which are defined constructively. Specifically, $\mathcal{C}$ consists of*

- *The cocones formed by datatypes and their generators, for those datatypes which are defined constructively using generators.*
- *The cocones formed by the morphisms indicating each distinct element of an enumerated type.*

$\mathcal{C}$ is used to identify those datatypes in which the individual elements must remain distinct. In Section 3.5.3, we referred to a common style of specification in which some datatypes are defined constructively using generators. These datatypes are typically defined within the data universe, and any implementation should demonstrate the no-confusion and no-junk properties with respect to these. The no-confusion property will ensure that each successive application of the generator function produces a new element of the datatype, while the no-junk property ensures that every element of the datatype is expressible using the generator functions. In Section 4.3.5 we show how $\mathcal{C}$ can be used to ensure that these properties exist in any implementation. Because defining these datatypes is the only way in which we use $\mathcal{C}$, every cocone in this family will be a coproduct of ones.

We note explicitly that the cocone representing the `state` datatype and its generators `init` and `next` for any component is included in $\mathcal{C}$. This will be of importance when we consider axioms relating to state, both in this chapter and Chapter 5. Not all specification languages use these generators to define a state datatype, but the underlying concept of a 'next' state is common amongst all languages we examine.

4.3.2 CCF and EA sketches

As mentioned earlier, $\mathcal{C}$ is used to identify those datatypes in which individual elements must remain distinct. These include datatypes such as the naturals, defined constructively using generators in the majority of CoFI languages. Many of these datatypes are required to be, like the naturals, countably infinite sets. That is, there are a countably infinite number of distinct elements (produced by successive applications of a generator function) in each of these datatypes. The cocone representing this datatype and its generators will then also be a countably infinite coproduct.

The necessity of representing these datatypes categorically explains the difference between the constructions of a CCF and EA sketch, and the need for the former. A finite coproduct — such as those permitted in the EA sketch concept — could not represent these datatypes. However, it is worth noting that specification languages do not, in practice, implement countably infinite sets for datatypes due to computational limits. Thus, it would be possible to represent an implementation of a specification language using EA sketches rather than CCF sketches. Nevertheless, we choose to use the more permissive CCF sketches of Definition 4.1 for theoretical precision.

When specifying a Rosetta system, we do not necessarily constrain this system and its components to demonstrate final behaviours only. Consequently, the `state` datatype for each component must consist of a countably infinite number of distinct elements. This is summarised in the following remark.

Remark 4.8. *For any component `fi` the `fi-state` datatype and its generators `fi-init` and `finext` (introduced for Rosetta in Appendix A) are represented by a cocone in $\mathcal{C}$ consisting of a countably infinite coproduct of ones.*

If we are examining a component in which all the behaviours consist of at most n state transitions, the `state` datatype can instead be represented by a coproduct of n ones.

We can draw a parallel between the theories of Section 3.3 and the CCF sketches $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ representing a particular system or component. The nodes and edges in G correspond to the signature Σ, while the constraints represented in $\mathcal{D}$ are equivalent the sentences E^*. The cones and cocones in $\mathcal{L}$ and $\mathcal{C}$ then provide extra information about datatypes within the data universe, which theories fail to include.

4.3.3 Associating a Category with a CCF Sketch

Given any EA sketch $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$, existing work [52] provides a means of generating an associated category. We can apply this to system specifications and CCF sketches as follows.

Definition 4.9. *For any CCF sketch $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ representing a system, the* system category $\boldsymbol{C}$ *is obtained from the free category upon G, but contains all finite limits and countable coproducts and is subject to the following restrictions:*

- *Images of pairs of paths in $\mathcal{D}$ are commutative diagrams in $\boldsymbol{C}$*
- *Images of cones in $\mathcal{L}$ are limit cones in $\boldsymbol{C}$*
- *Images of cocones in $\mathcal{C}$ are colimit cocones in $\boldsymbol{C}$*

This category $\mathbf{C}$ is often termed the classifying category [9], and is technically a model of the sketch. In future definitions, we will refer to a category $\mathbf{C}$ generated in this way as the category *presented* by the CCF sketch. $\mathbf{C}$ therefore represents all the information that is contained in the system specification.

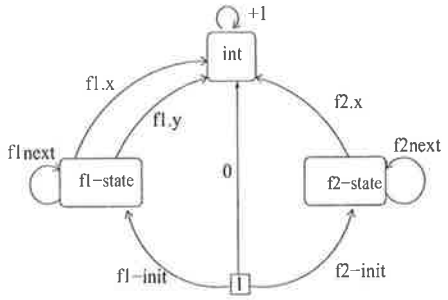

Figure 4.1: Part of the category C generated by the graph G for Example 1

4.3.4 An Example Sketch

Figure 4.1 shows the graph G for Example 1 of Section 4.3.1, although for space considerations we have omitted some elements of the data universe. $\mathcal{D}$ consists of the pairs determined by the axioms present in each component:

$((f1.y \circ f1next), (+1 \circ f1.y))$ (facet f1 axiom T1)
$((f2.x \circ f2next), (+1 \circ f2.x))$ (facet f2 axiom L0)
etc

as well as the axioms defining the data universe.

As we noted in Clarification 4.5, $\mathcal{D}$ contains no axioms which reflect the fact that the variable x is shared between components `f2` and `f1`. The issue of consistency (that these are not separate variables and so must have the one common value in each given state) is discussed in Section 4.5.2. This, however, motivates the names $f1.x, f2.x$ of the edges in G representing these variables.

In this example, $\mathcal{L}$ consists of the cones defining product datatypes, such as `int` $\times$ `char`. The cocones in $\mathcal{C}$ represent the abstract datatypes `f1-state` and `f2-state`, as well as the datatype `int`. As mentioned earlier, due to space considerations we have not shown the entire graph in Figure 4.1. In reality, this graph would also include other generator functions for the integers, as well as all product datatypes. The system category $\mathbf{C}$ can also be represented by Figure 4.1.

4.3.5 The Models of a Specification

A model of a specification is any mapping of elements of the specification to sets and functions. A consistent model is one for which the mapping obeys all the constraints of the specification. Models can take many different forms, depending upon the framework used. For example, in Chapter 3 we use algebras as models of Rosetta facets. Other types of models include coalgebras and finite

state automata. However, when identifying the models of a specification here, we will use **Set**, the category of sets.

Definition 4.10. *Given a specification associated with a system category* $\mathbf{C}$*, a* model *of this specification is a functor* $D : \mathbf{C} \rightarrow$ ***Set*** *preserving finite limits and countable coproducts.*

This functor $D : \mathbf{C} \rightarrow \mathbf{Set}$ will be termed a *system model*, and provides an implementation in **Set** of the system associated with $\mathbf{C}$. An underdefined system may have several valid system models D. For example, if the value of a integer variable x is undefined in the initial state, then there are a countably infinite number of valid models D, each of which maps x (in the initial state) to a different integer. A single model D according to Definition 4.10 describes a trace, or a value for each variable in each state, for every component within the system.

Because a system model D preserves countable coproducts, it will map a coproduct of n ones in $\mathbf{C}$ to a set consisting of n distinct elements. More generally, D will map a countable coproduct of ones in $\mathbf{C}$ to a countable set. As a consequence, a system model D maps any datatype represented by a colimit cocone in $\mathcal{C}$ to a set consisting solely of elements defined by the generator functions of this datatype. The following remark highlights one important consequence of this, which will later be used to implement axioms based upon state.

Remark 4.11. *D maps each object `fi-state` representing the state of a component `fi` in $\mathbf{C}$ to a countable set, ordered according to the action of D on the `fi-init` and `finext` functions defining the state datatype of `fi`.*

As mentioned before, if two paths are included in $\mathcal{D}$, then the morphisms corresponding to these paths are equal in the system category $\mathbf{C}$. The functor $D : \mathbf{C} \rightarrow \mathbf{Set}$ then preserves this equality, mapping this one morphism to a single set-valued function in **Set**. Thus, the equalities of the specification are preserved within the image of $\mathbf{C}$ in **Set**. In algebraic terms, we may think of the functor D as the denotational function, while the category $\mathbf{C}$ represents the signature. Given a list of axioms which form the presentation, the family of system models D is then analogous to the variety of algebras over this presentation. However, the advantage of this categorical approach over the algebraic method is that there is no need to list the axioms separately to form this presentation. Instead, they are already contained within the equalities present in $\mathbf{C}$.

4.4 State Specification Using A CCF Sketch

In this chapter, we have so far shown how to use a system category $\mathbf{C}$ to represent a system specification, while a functor $D : \mathbf{C} \rightarrow \mathbf{Set}$ acts as a model of this specification. Here, $\mathbf{C}$ and D together describe the different values that variables have throughout a simulation trace. However, in order to analyse system behaviour, CCF must support the identification of individual states of the entire

system. This should be done in a manner which enables us to identify states of individual components, thereby obtaining each component's perspective on the system. We will use *views* — from database theory — to achieve this, defining a category $\mathbf{C}_V$ to represent the data visible in a single state. A state will then be a functor from $\mathbf{C}_V$ to **Set**, mapping data elements to values.

4.4.1 The CCF Sketch Representing A State

Given a system category $\mathbf{C}$ representing a system, we can denote a state of this system by defining a subsketch identifying a subcategory of $\mathbf{C}$. This subsketch is similar to a database view, and identifies elements of $\mathbf{C}$ which represent system data visible in a single state. If we restrict the discussion to the practical implementation of specification languages — and so use EA sketches rather than CCF sketches, as discussed in Section 4.3.2 — then we may define such a subsketch precisely as a view. However, this additional theoretical complexity is not required for our purpose.

Definition 4.12. *An CCF sketch* $(G_V, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$ *representing a single state of a system is a tuple consisting of:*

- *A directed graph* G_V *whose nodes represent datatypes of the system and whose edges represent functions, constants, and single values of variables*
- *A family* $\mathcal{D}_V$ *of some of the pairs of paths in* G_V *with common source and target, representing the axioms within the data universe of the specification*
- *A family* $\mathcal{L}_V$ *of some of the cones in* G_V
- *A family* $\mathcal{C}_V$ *of some of the cocones in* G_V

We discuss the construction of this sketch in more detail below.

The Graph G_V Representing an Individual State

Clarification 4.13. *The graph* G_V *of Definition 4.12 is comprised of*

- *Nodes which are identical to the nodes of* G.
- *Edges which represent all functions, constants and variables of each component (except the* `init` *and* `next` *functions), where variables are implemented as functions with a null domain.*

G_V does not include edges representing the `init` and `next` functions which are part of the `state` datatype for each component. It does, however, include the variable `current` of type `state` for each component.

G_V is similar to the graph G described in Clarification 4.4, with a few notable exceptions. Firstly, variables in G_V are represented as functions with a null domain, rather than functions dependent on state. This is in accordance with the notion that in each state a variable has a single value. That is, a functor from the category presented by $(G_V, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$ to **Set** will effectively associate each variable with a single value. Secondly, G_V does not include the functions `init` and `next` which generate the `state` datatype for any component. In Section 4.6.1 we observe that their presence in G_V would permit us to distinguish states which should be identical in a Basic State system, hence they must be omitted. Finally, the variable `current` for each component is included in G_V, although this was lacking from G. This, as we will discuss in Section 4.5.1, will be used to implement axioms based upon state. Figure 4.2 depicts part of the graph G_V for the code of Example 1.

Data Universe Equalities and the Family $\mathcal{D}_V$

Clarification 4.14. *The family $\mathcal{D}_V$ consists of pairs of paths in G_V, representing those pairs of functions or constants which are within the data universe and constrained to be equal by data universe axioms.*

The equalities within $\mathcal{D}_V$ are also represented in the family $\mathcal{D}$ described in Clarification 4.5. However, $\mathcal{D}$ also represents those system axioms which do not define the data universe (in Rosetta, those axioms which are within facets but *not* within domains), which $\mathcal{D}_V$ does not. This difference is because we intend to use CCF to consider inconsistent states, and the inclusion of consistency axioms within $\mathcal{D}_V$ would preclude this.

Limits $\mathcal{L}_V$ and colimits $\mathcal{C}_V$

Construct 4.15. *The families $\mathcal{L}_V$ and $\mathcal{C}_V$ represent cones and cocones of G_V which possess limit and colimit properties respectively. $\mathcal{L}_V$ is identical to $\mathcal{L}$ of Construct 4.6 and $\mathcal{C}_V$ is identical to $\mathcal{C}$ of Construct 4.7, except that $\mathcal{C}_V$ lacks the cocones representing the* `state` *datatypes for the components.*

$\mathcal{C}_V$ therefore contains cocones representing each constructively specified datatype (that is, those which are specified constructively using generator functions as discussed in Section 3.5.3), with the exception of the `state` cocones. The datatype `fi-state` (for any component `fi`) is the only constructively specified datatype which does not originate in the data universe. The cocones representing this datatype for each component cannot be included in $\mathcal{C}_V$, as the graph G_V does not contain the `init` morphism which is part of each such cocone.

4.4.2 Associating a Category with a State

As in Section 4.3.3 we generate the the category presented by the tuple $(G_V, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$ from the free category upon G, but subject to the usual restrictions upon the images of elements of the tuple.

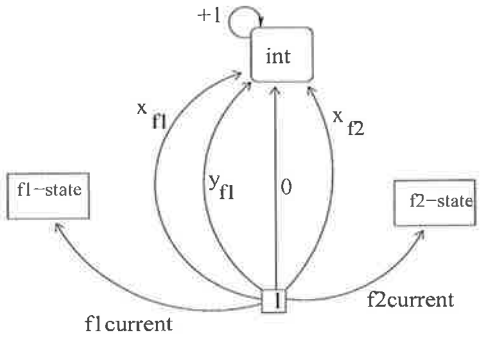

Figure 4.2: Part of the abstract state category for the code in Example 1

Definition 4.16. *The* abstract state category *is the category $\mathbf{C}_V$ presented by the CCF sketch $(G_V, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$ of Definition 4.12.*

The abstract state category represents elements of the system within a single state.

Figure 4.2 shows the category $\mathbf{C}_V$ for the code of Example 1. Again, for space considerations we do not include every datatype and function which would be defined in the data universe. The change in the naming of morphisms in $\mathbf{C}_V$ from their equivalent morphisms in $\mathbf{C}$ makes it easier to distinguish which category we refer to when discussing a morphism representing a variable. The abstract state category $\mathbf{C}_V$ is then used to define the state of a system as below.

4.4.3 Models Representing State

Definition 4.17. *A model of the abstract state category is a functor $R : \mathbf{C}_V \rightarrow$ **Set** which preserves finite limits and countable coproducts. This will be referred to as a* state *of the system.*

A state is a functor $R : \mathbf{C}_V \rightarrow \mathbf{Set}$. R serves to map the elements of $\mathbf{C}_V$ to their values in the state which R represents. There are, of course, some conditions which R must satisfy in order for it to be a consistent state. For example, it must map variables to values which satisfy the system constraints embodied in $\mathbf{C}$. In the following section, we define precisely what it means for a state to be consistent, and identify those inconsistencies which might arise in a single state.

4.5 Consistency of An Individual State

In this section we will consider the ways in which an individual state might be inconsistent. This does not include a full discussion of inconsistent *behaviour*,

which is deferred to Sections 4.6 and 4.7. There are two classes of inconsistency which can affect an individual state. The first of these we term *component-wise inconsistency*. Inconsistencies of this form arise due to conflicts between the constraints of a single component. That is, this type of inconsistency arises when a component lacks internal consistency. For example, the initial state of the following component will always demonstrate component-wise inconsistency.

```
facet inconsistent:: state-based
  x: int;
  begin
   T0: x@s0=5;
   T1: x@s0=6;
end inconsistent
```

This type of inconsistency arises from conflicting requirements documents, or errors made by a single designer specifying the component in question. These inconsistencies can be found by examining each component in isolation, and can generally be rectified without needing to alter the specification of any other components within the system.

The second form of inconsistency affecting an individual state is termed *interaction-wise inconsistency*. These inconsistencies arise when a previously-consistent component is placed within a system containing constraints which would affect this component. For example, the following components `f1` and `f2` share a variable x and cannot co-exist in a system if the states are to demonstrate interaction-wise consistency.

```
facet f1:: statebased
  public x: int
  begin
   T0:x = 5;
end f1
```

```
facet f2:: statebased
  begin
   L0:x = 6;
end f1
```

These inconsistencies are due to the sharing of information between components. They arise from conflicts between the constraints the two components place on the shared information.

Definition 4.18. *A system state is* component-wise consistent *if in this state every component, when considered in isolation, satisfies its constraints.*

Definition 4.19. *A system state is* interaction-wise consistent *if information shared between multiple components has the same value when observed from any component.*

Component-wise and interaction-wise inconsistencies can exist independently in any given system.

4.5.1 Component-wise Consistency

In Section 4.4, we introduced a functor $R : \mathbf{C}_V \rightarrow \mathbf{Set}$ representing the state of a system. In order for this to represent a component-wise consistent state, R must map variables to values which satisfy the constraints within each component. The structure of the categories of CCF is such that these constraints are represented in the system category $\mathbf{C}$ in the form of equalities of morphisms. Thus, for a state R to satisfy these axioms, it is sufficient that R factor through $\mathbf{C}$ in such a way that the constraints are applied to the relevant variables.

We describe this formally by creating the following functor. This helps define the action of any state functor $R : \mathbf{C}_V \rightarrow \mathbf{Set}$.

Definition 4.20. *The* state identification functor *is a functor* $\bar{R} : \boldsymbol{C}_V \rightarrow \boldsymbol{C}$. *It is* valid *if the following conditions are satisfied*

1. $\bar{R}$ *maps any object representing a datatype, or any morphism in* $\boldsymbol{C}_V$ *representing a function application, function or constant, to the object or morphism of* $\boldsymbol{C}$ *representing this same datatype, constant, function or function application.*

2. *For each component* `fi` *and for any variable* x *of this component, if*

$$\bar{R}(ficurrent) = finext^k \circ fi\text{-}init$$
$$\text{then } \bar{R}(x_{fi}) = fi.x \circ finext^k \circ fi\text{-}init$$

   where x_{fi} *and* $fi.x$ *are the morphisms representing the variable* x *of the component* `fi` *in categories* $\boldsymbol{C}_V$ *and* $\boldsymbol{C}$ *respectively.*

These state identification functors $\bar{R}$ relate the morphisms in the abstract state category $\mathbf{C}_V$ to the morphisms in the system category $\mathbf{C}$. $\bar{R}$ may also be defined as a *sketch morphism*. We will use these functors to describe how states R factor through $\mathbf{C}$.

The first condition for validity in Definition 4.20 states that a valid $\bar{R}$ takes objects in $\mathbf{C}_V$ (representing datatypes) to the objects representing these datatypes in $\mathbf{C}$. This also applies to any morphisms in $\mathbf{C}_V$ representing functions and constants (although not variables). More specifically, $\bar{R}$ acts in the same way as an inclusion functor upon the data universe, the datatype `fi-state` for each component `f1`, the terminal node, and the morphisms representing functions and constants. These objects are static throughout the progression of a simulation.

The second condition for validity in Definition 4.20 explains the nomenclature chosen for this functor $\bar{R}$. Each $\bar{R}$ maps any morphism x_{fi} in $\mathbf{C}_V$ representing a variable x of the component `fi` to a morphism in $\mathbf{C}$ which represents this variable x after a certain number of state transitions of `fi`. In this way, each $\bar{R}$ identifies the number of state transitions which each component has undergone.

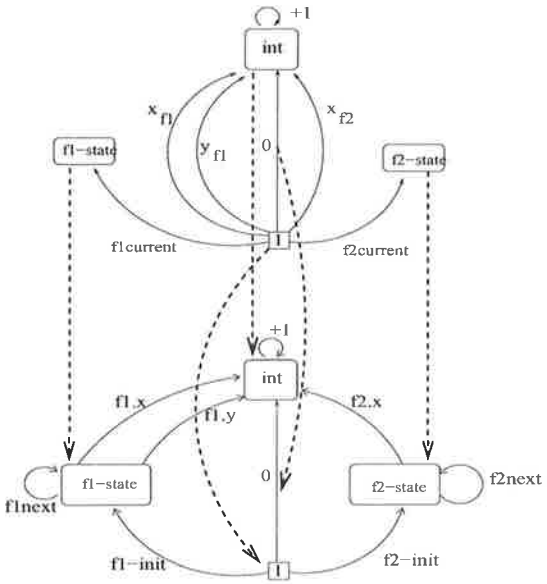

Figure 4.3: The operation of $\bar{R}$ on parts of C_V

That is, if the action of $\bar{R}$ indicates that a component `fi` has undergone k state-transitions, then $\bar{R}$ maps the morphism x_{fi} in $\mathbf{C}_V$ (representing the variable x in `fi`) to the morphism in $\mathbf{C}$ which represents the variable x in `fi` after k state transitions. We will later show how $\bar{R}$ can be used to construct a state R which is observable after this number (k) state transitions of `fi`. This second condition in Definition 4.20 also ensures that the axioms in $\mathbf{C}$ constraining certain variables will be applied in any state R to the correct variables.

Figure 4.3 shows the action of a state identification functor $\bar{R}$ upon the categories associated with Example 1 in Section 4.3.1. We note that under the conditions for validity in Definition 4.20, a state identification functor will be well-defined and functorial. This is because any equalities in the categories $\mathbf{C}$ and $\mathbf{C}_V$ (differentiating these categories from the free categories upon graphs G and G_V) are obtained from the families $\mathcal{D}_V$ and $\mathcal{D}$. In Section 4.4 we showed $\mathcal{D}_V \subseteq \mathcal{D}$, meaning any commutativity which exists in $\mathbf{C}_V$ also exists in $\mathbf{C}$.

State identification functors also allow us to express the factorization of a state R through the system category $\mathbf{C}$, as the following theorem describes.

Theorem 4.21. *$R : \boldsymbol{C}_V \to \boldsymbol{Set}$ represents a component-wise consistent state if and only if there exists a system model D and valid state identification functor $\bar{R} : \boldsymbol{C}_V \to \boldsymbol{C}$ such that $R = D \circ \bar{R}$.*

Proof. We consider a constraint $(t1 = t2)$ in the specification, and prove that this is satisfied in a state R which meets the above conditions. From the discussion following Definition 4.2, we know that terms $t1$ and $t2$ will be either

1. functions or constants
2. variables in the initial state
3. variables
4. function applications to any of the above

Because we are examining a single state R, we will assume that neither $t1$ nor $t2$ contains an application of the `next` function. Throughout this proof, we use Definition 4.9 and the points emphasised in Clarification 4.5 to assert that there is an equality $m1 = m2$ in $\mathbf{C}$, where $m1$ and $m2$ are the morphisms in $\mathbf{C}$ which are the image of those paths in G representing $t1$ and $t2$. We let $v1$ and $v2$ be the morphisms in $\mathbf{C}_V$ representing the terms $t1$ and $t2$ (noting that points (2) and (3) are the same when represented in $\mathbf{C}_V$, since $\mathbf{C}_V$ does not refer to multiple states). Since we are considering only those terms which do not contain the `next` function, these morphisms $v1$ and $v2$ are guaranteed to exist.
1). We first consider a constraint $(t1 = t2)$ in the specification, where $t1$ and $t2$ are either functions or constants. Definition 4.20 tells us that $\bar{R}$ acts as the identity functor on constants and functions. That is, we can deduce

$$\bar{R}(v1) = m1 = m2 = \bar{R}(v2)$$

2). We now consider a constraint $(t1 = t2)$ in the specification, where one or both of $t1$ or $t2$ are variables in the initial state of the appropriate component. Without loss of generality, we suppose $t1$ is a variable x of some component `fi` in the initial state of `fi`, while $t2$ is a variable y of `fi` in the initial state. Here, the morphisms $v1$ and $v2$ are respectively x_{fi} and y_{fi}, while the morphisms $m1$ and $m2$ are respectively $fi.x \circ fi\text{-}init$ and $fi.y \circ fi\text{-}init$. By Definition 4.20, if $\bar{R}(ficurrent) = fi\text{-}init$ then $\bar{R}(v1) = fi.x \circ fi\text{-}init$ and $\bar{R}(v2) = fi.y \circ fi\text{-}init$. Thus, if $\bar{R}(ficurrent) = fi\text{-}init$ we have

$$\bar{R}(v1) = m1 = m2 = \bar{R}(v2)$$

3). We now consider a constraint $(t1 = t2)$ in the specification, where $t1$ and $t2$ are variables of some component `fi`. Here, $m1$ and $m2$ are the morphisms $fi.x$ and $fi.y$, for example, while $v1$ and $v2$ are the morphisms x_{fi} and y_{fi}. Supposing the component `fi` has undergone k state transitions, by Definition 4.20 we know

$$\bar{R}(v1) = m1 \circ finext^k \circ fi\text{-}init$$

and

$$\bar{R}(v2) = m2 \circ finext^k \circ fi\text{-}init$$

Since we assumed $m1 = m2$, we can then conclude

$$\bar{R}(v1) = \bar{R}(v2).$$

4). Finally, a constraint $(t1 = t2)$ may contain one or more function applications to terms of type 1) - 3). By Definition 4.20, we know that $\bar{R}$ acts as the identity on functions. Since function application is modelled by composition of the morphisms in $\mathbf{C}$ representing the function and its arguments, if the required conditions for the equalities in parts 1) - 3) of this proof hold, we can conclude

$$\bar{R}(v1) = m1 = m2 = \bar{R}(v2)$$

when either $t1$ or $t2$ represents a function application. This means that for any system model $D : \mathbf{C} \rightarrow \mathbf{Set}$ and for any constraint $(t1 = t2)$ the equality

$$D \circ \bar{R}(v1) = D \circ \bar{R}(v2)$$

holds. Thus, for any $R = D \circ \bar{R}$, this functor R will represent a component-wise consistent state, meaning the system constraints will be satisfied in the state R.

To prove the other direction, let $R : \mathbf{C}_V \rightarrow \mathbf{Set}$ be a component-wise consistent state, according to Definition 4.18 and achievable after some k state transitions of each component. Furthermore, let v_1 and v_2 be morphisms within $\mathbf{C}_V$ for which $R(v_1) = R(v_2)$.
This means that either

- there is a system constraint $t_1 = t_2$ which applies in state R, where t_1 and t_2 are variables or functions represented in $\mathbf{C}_V$ by morphisms v_1 and v_2 (since R is component-wise consistent, this system constraint must be satisfied, i.e. $R(v_1) = R(v_2)$)
- t_1 and t_2 are underconstrained variables or functions, represented in $\mathbf{C}_V$ by morphisms v_1 and v_2.

In the first case, for any valid state identification functor $\bar{R}$ representing a state achievable after k state transitions, the equality

$$\bar{R}(v_1) = \bar{R}(v_2)$$

will hold. This is because $\bar{R}(v_1)$ and $\bar{R}(v_2)$ represent — in $\mathbf{C}$ — the system terms t_1 and t_2 after k state transitions, and by definition system constraints such as $t_1 = t_2$ are represented as commutative diagrams within $\mathbf{C}$.

In the second case, there exists a system model $D : \mathbf{C}_V \rightarrow \mathbf{Set}$ for which

$$D \circ \bar{R}(v_1) = D \circ \bar{R}(v_2)$$

for a valid state identification functor $\bar{R}$ representing a state achievable after k state transitions. This is because the set of system models consists of all the functors $D : \mathbf{C} \rightarrow \mathbf{Set}$ which preserve finite limits and countable coproducts. As we can see from the tuple $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ introduced in Section 4.3.1, the images of variables and functions in $\mathbf{C}$ (so particularly, the morphisms $\bar{R}(v_1)$ and $\bar{R}(v_2)$) will not be part of any cone or cocone in $\mathbf{C}$. This means that it is possible for a functor $D : \mathbf{C} \rightarrow \mathbf{Set}$ to satisfy the equality above and still preserve finite

limits and coproducts. Hence, because the set of system models consists of all possible functors $D : \mathbf{C} \rightarrow \mathbf{Set}$, there will be at least one such functor satisfying this equality.

That is, in either case there exists a functor $R' = D \circ \bar{R}$ such that

$R'(v_1) = R'(v_2)$ for any morphisms v_1, v_2 in $\mathbf{C}_V$ such that $R(v_1) = R(v_2)$ (**)

From Definition 4.17 we know that R preserves finite limits and countable coproducts of $\mathbf{C}_V$, as does any $R' = D \circ \bar{R}$. Moreover with the exception of the objects representing the `fi-state` datatypes, every object in $\mathbf{C}_V$ is a coproduct object. This means that both R and R' map these objects to countable ordered sets. The objects in $\mathbf{C}_V$ representing `fi-state` datatypes are mapped by R to any set (where $R(ficurrent)$ is an injection from the singleton set into this set), and by R' to an infinite coproduct of ones.
It is clear from this and (**) that we can define an isomorphism between R and R' by means of set functions acting on countable sets. That is, up to isomorphism, we have expressed R as the composition of a state identification functor $\bar{R}$ and a system model D.

□

4.5.2 Interaction-wise Consistency

A functor $R : C_V \rightarrow \mathbf{Set}$ may represent a state which is component-wise consistent and yet fails to be interaction-wise consistent. This is because, while R factors through the system category $\mathbf{C}$, this category $\mathbf{C}$ does not contain any equalities which would ensure that all shared information is interpreted identically by each component. For example, in the system category corresponding to Example 1 (Section 4.3.1), there are no equalities which enforce that the value of $f1.x$ is equal to the value of $f2.x$ at any point. The advantage of this separation between the different types of consistency is that it allows us to examine inconsistent systems using CCF, a topic to which we return in Section 4.7.

We can model interaction-wise consistency using CCF by defining an *interaction consistency category* $\bar{\mathbf{C}}_V$. This category will represent the data which is visible in a single state of the system, just as the abstract state category $\mathbf{C}_V$ does. However, the interaction consistency category will also contain equalities which ensure that at any point during a trace, a shared variable will have the same value when seen by any component. By ensuring a state R factors through $\bar{\mathbf{C}}_V$, we will therefore guarantee interaction-wise consistency of R.

The CCF Sketch Representing the Interaction Consistency Category

As we have done for all other categories comprising CCF, we generate $\bar{\mathbf{C}}_V$ from a CCF sketch $(G_V, \mathcal{D}_V \bigcup \bar{\mathcal{D}}_V, \mathcal{L}_V, \mathcal{C}_V)$. The elements of this tuple are discussed below.

Definition 4.22. *A CCF sketch representing a single interaction-wise consistent state of a system is a tuple* $(G_V, \mathcal{D}_V \bigcup \bar{\mathcal{D}}_V, \mathcal{L}_V, \mathcal{C}_V)$ *consisting of*

- *The directed graph* G_V *of Clarification 4.13, used in Definition 4.12 to generate the abstract state category of this system*
- *A family* $\mathcal{D}_V \bigcup \bar{\mathcal{D}}_V$ *of pairs of paths in* G_V *with common source and target, where* $\mathcal{D}_V$ *is the family of Clarification 4.14 and* $\bar{\mathcal{D}}_V$ *represents constraints which enforce equality of shared variables*
- *The family* $\mathcal{L}_V$ *of cones of* G_V *detailed in Construct 4.15 and used in Definition 4.12 to generate the abstract state category*
- *The family* $\mathcal{C}_V$ *of cocones of* G_V *detailed in Construct 4.15 and used in Definition 4.12 to generate the abstract state category*

The elements $G_V, \mathcal{D}_V, \mathcal{L}_V$ and $\mathcal{C}_V$ of the CCF sketch tuple defining the interaction consistency category are precisely those of the tuple $(G_V, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$ associated with the abstract state category C_V. The family $\bar{\mathcal{D}}_V$ represents those constraints which ensure a shared variable has a single value at any time.

Clarification 4.23. *The family* $\bar{\mathcal{D}}_V$ *consists of pairs of paths in* G_V *with a common source and target. These will be those pairs* (e_1, e_2) *where* e_1 *and* e_2 *represent a shared variable as observed by two different components.*

For example, for a variable x shared between components `f1` and `f2`, $\bar{\mathcal{D}}_V$ includes the pair (x_{f1}, x_{f2}). Constructing $\bar{\mathcal{D}}_V$ in this way means that there can be no intersection between the families $\bar{\mathcal{D}}_V$ and $\mathcal{L}_V$ or $\mathcal{C}_V$. That is, the cones and cocones in the interaction consistency category presented by this CCF sketch will be identical to the cones and cocones in the abstract state category.

Generating the Interaction Consistency Category

The CCF sketch of Definition 4.22 presents a category $\bar{\mathbf{C}}_V$, the interaction consistency category.

Definition 4.24. *The* interaction consistency category *is the category* $\bar{\mathbf{C}}_V$ *presented by the CCF sketch* $(G_V, \mathcal{D}_V \bigcup \bar{\mathcal{D}}_V, \mathcal{L}_V, \mathcal{C}_V)$ *of Definition 4.22.*

The interaction consistency category $\bar{\mathbf{C}}_V$ is structurally identical to the abstract state category $\mathbf{C}_V$ of Definition 4.16, save for the addition of the equalities in $\bar{\mathcal{D}}_V$. This similarity can be formalised by defining the *quotient functor*

$$I : \mathbf{C}_V \to \bar{\mathbf{C}}_V$$

That is, for morphisms x_{f1} and x_{f2} in $\mathbf{C}_V$ which represent a shared variable x as seen by components `f1` and `f2` respectively, the equality

$$I(x_{f1}) = I(x_{f2})$$

will apply in $\bar{\mathbf{C}}_V$. We will make use of similar quotient functors throughout this thesis.

The following theorem describes how the interaction-wise consistency category is used in determining the consistency of a state.

Theorem 4.25. *$R : \boldsymbol{C_V} \rightarrow$ **Set** is an interaction-wise consistent state if there exists a functor $Y : \bar{\boldsymbol{C}}_V \rightarrow$ **Set** such that $R = Y \circ I$, where I is the quotient functor.*

Proof. Let $x_{f1}, x_{f2} : t \rightarrow \texttt{dtype}$ be morphisms in $\mathbf{C}_V$ representing the variable x of components `f1` and `f2`, where this variable is shared between these components. Then by the definition of the quotient functor I we know

$$I(x_{f1}) = I(x_{f2})$$

and therefore

$$Y \circ I(x_{f1}) = Y \circ I(x_{f2})$$

If $R = Y \circ I$, then this means R represents an interaction-wise consistent state. □

4.5.3 Complete Consistency of a State

To be consistent, a state must be both component-wise and interaction-wise consistent. Moreover, it must comply with additional designer expectations, which are primarily related to the constructive specification of certain datatypes. The following lemma describes what criteria a functor $R : \mathbf{C}_V \rightarrow \mathbf{Set}$ must satisfy if it is to represent a consistent state.

Lemma 4.26 (Consistency Lemma). *$R : \boldsymbol{C_V} \rightarrow$ **Set** is* consistent *in the senses of Definitions 4.18 and 4.19 and* valid *iff*

1. *$R = D \circ \bar{R}$ for some valid state identification functor $\bar{R}$ and system model D*
2. *$R = Y \circ I$ for the quotient functor $I : \boldsymbol{C_V} \rightarrow \bar{\boldsymbol{C}}_V$, and some functor $Y : \bar{\boldsymbol{C}}_V \rightarrow$ **Set***
3. *R preserves finite limits and countable coproducts*

The first two conditions of this lemma state that, to be consistent, a state R must be both component-wise consistent (1) and interaction-wise consistent (2). For ease of notation, we will henceforth refer to a component-wise consistent state as one which *factors through* the system category $\mathbf{C}$. The third condition ensures that the limits and coproducts which contain information about the system are preserved. For example, the datatypes specified constructively and intended to possess the no-junk and no-confusion properties are represented in $\mathbf{C}_V$ as coproducts of ones. $R : \mathbf{C}_V \rightarrow \mathbf{Set}$ must then preserve these coproducts if the no-junk and no-confusion properties are to hold.

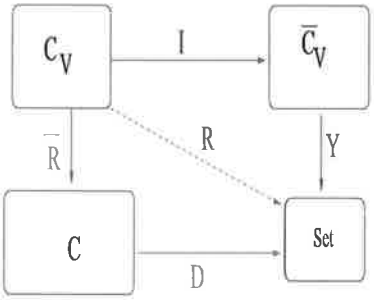

Figure 4.4: This diagram must commute for R to represent a consistent state

The requirement that R preserve these limits and coproducts also ensures that if a component specification further constrains these datatypes (which all originate in the data universe and hence should not be altered by components) then states of any system containing this specification will be invalid. This is because the cocone in $\mathbf{C}_V$ representing the datatype in question is not affected by the presence of this constraint within a component specification, since $\mathbf{C}_V$ only contains a representation of the constraints originating in the data universe. However, the cocone in $\mathbf{C}$ representing this datatype is affected by this constraint in the component specification. That is, the colimit cocones of $\mathbf{C}$ and $\mathbf{C}_V$ representing this datatype will no longer be identical. When R is factored through $\mathbf{C}$, this constraint is applied to the image of the cocone in $\mathbf{C}_V$, meaning that R cannot preserve limits and coproducts as required by Lemma 4.26.

These conditions on R can be expressed diagrammatically. R will be a valid and consistent state when the diagram in Figure 4.4 commutes, and R preserves finite limits and countable coproducts. To demonstrate the utility of this framework and provide a template for identifying and resolving inconsistencies, Section 4.7 establishes examples of several different types of inconsistency. We also discuss therein some techniques for resolving these inconsistencies. However, to provide a motivation for this taxonomy of inconsistencies, we present a small example below in which component-wise and interaction-wise inconsistency appear.

4.5.4 An Inconsistency Example

The following specification demonstrates how some different types of inconsistency might arise in a system. This is a Rosetta specification of the light level for safety lights in a building which must be constantly illuminated. The facet `switchmethod` accepts as input from the user an argument `setting` giving the level of light required, which must be 1, 2 or 3. A function `transform` (definition not shown) uses this to decide how many lights should be illuminated, this being output via the parameter `lightnumber`. However, this requires a certain

amount of power, dependent upon the value of the `setting` parameter. As seen below, our user has mistakenly added two axioms (T0, T2) which conflict and so should be identified as an inconsistency.

```
facet switchmethod(setting: input [1, 2, 3],
                   lightnumber: output int):: state-based
  public int power1;
  begin
   T0: power1' = 2*setting;
   T1: lightnumber' = transform(setting);
   T2: power1' = 3*setting;
end switchmethod
```

This part of the building also contains an alarm circuit which draws power. For safety reasons, the total power drawn from the alarm and lights must be constant. We represent this by the following component, in which `power1` represents the power used by the lights (visible via the `public` mechanism from `switchmethod`) and `power2` represents the power used by the alarm.

```
facet powerreq:: state-based
  const int powerconstant = 10;
  begin
   A0: power1 + power2 = powerconstant;
end powerreq
```

However, the designer of the alarm circuit has also constrained the power requirements of the alarm to be dependent upon an input value from the user, representing how much coverage this alarm should offer. This value, `alarmon`, can be seen in the `alarmreq` facet below, and corresponds to the number of sectors in which the alarm is active.

```
facet alarmreq(input alarmon: int):: state-based
  public int power2;
  begin
   L0: power2 = alarmon;
end alarmreq
```

We can examine a state R of this system, to identify any inconsistencies which might be present in the specification. If we assume that R preserves limits and coproducts as required by the conditions of Lemma 4.26 (and therefore that the integers are a countably infinite set, ordered by the + and - functions and constant 0), then the presence of axioms `switchmethod.T2` and `switchmethod.T1` indicates an inconsistency. Specifically, these axioms demonstrate that there is no valid state R which can be factored through the system category $\mathbf{C}$.

In more detail, if we suppose that R factors through $\mathbf{C}$, then the axioms $T0$ and $T2$ (which are represented as equalities within C) prevent R from preserving limits and coproducts as required by the conditions of Lemma 4.26. That is, these axioms can only hold if, for any given integer i, $2 * i = 3 * i$. As a result, R will be invalid. Section 4.7 explains the effect of this type of inconsistency in more detail.

We can also detect potential inconsistencies which arise when components are placed together. For example, the constraints upon the `power2` variable and input `alarmon` may result in a conflict. This occurs if the user supplies certain values for `alarmon` (eg. 20). This is because there would then be no possible value of the `power1` variable in facet `switchmethod` which would allow the axiom `powerreq.A0` to be satisfied. More generally, we may say that there exist system models D which cannot exist as part of a commuting diagram as in Figure 4.4.

In Section 4.7 we discuss the different types of inconsistency in more depth. This allows us to take a systematic approach to the resolution of inconsistencies, such as those which arise in the `lightswitch` system. However, the majority of inconsistencies are not constrained to an individual state, but occur several times in the course of a behaviour. To adequately determine the effect of these, we must firstly relate states to each other and secondly define precisely what constitutes a *system behaviour*. The following section describes how to do this, using the category theoretic notions of CCF which have been introduced so far.

4.6 The Category of States

The behaviours observable in a consistent system are determined by the relationships between consistent states. In order to identify and resolve inconsistencies in behaviour, we require a formal framework where these relationships between states can be modelled explicitly. In this section, we use the categories of CCF to provide such a framework — namely, the category of states of a system. Morphisms in this category of states will represent behaviours of the system, which will allow us to examine both consistent and inconsistent behaviours in the same framework. We will make use of this ability in Section 4.7, where we present a taxonomy of inconsistencies.

For a given Basic State system, objects in the category **States** will be consistent states of this system, and morphisms will represent the Basic behaviours of this system. Of course, not every system is a Basic State system, and in Chapter 5 we discuss how to form the category of states of other types of system. However, in the rest of this chapter we restrict our attention to Basic State systems only.

Definition 4.27. *The category* ***States*** *is comprised of*

- *objects which are functors* $R : \mathbf{C}_V \rightarrow$ ***Set*** *which are consistent and valid in the sense of Definitions 4.18 and 4.19, and Lemma 4.26*
- *morphisms which are tuples* $(R, [R_i], R')$ *of objects in* ***States*** *which satisfy the State Ordering Property (given below as Property 4.28)*

A morphism r then describes the states observed throughout a system behaviour, although as we show later, not necessarily a *consistent* system behaviour.

4.6.1 Objects Within the Category of States

Objects of **States** are thus consistent states of the system under analysis, and equality of objects is equality of the values visible in each state. That is, two objects $R, R' : \mathbf{C}_V \rightarrow$ **Set** will represent equal states if and only if R and R' are naturally isomorphic as functors. One consequence of this definition of equality is that the differing values of the variable *ficurrent* for any component `fi` cannot distinguish two otherwise identical states. This is because $\mathbf{C}_V$ does not represent the functions `fi-init` and `finext` which impose an ordering on the datatype `fi-state` for any component `fi`. Lacking this ordering, it is always possible to define an isomorphism $\sigma : R(\texttt{fi-state}) \rightarrow R'(\texttt{fi-state})$ in **Set** for any component `fi`, such that

$$\sigma(R(ficurrent)) = R'(ficurrent)$$

In simpler terms, a consequence of the equality of Definition 4.27 is that the number of state transitions undergone by any component `fi` is not sufficient to distinguish otherwise identical states.

4.6.2 Morphisms Within The Category Of States

Equality of morphisms in **States** is defined as equality of the tuples, up to the conflation of identical consecutive objects in either tuple. This conflation ensures that behaviours in **States** are stuttering-insensitive. Composition of morphisms is then concatenation of the ordered lists $[R]_i$ which make up the tuple of each morphism. Thus, if $r = (R, [R]_i, R')$ and $r_2 = (R', [R']_j, R_2)$ are morphisms in **States**, the composition of these will be the morphism

$$r_2 \circ r = (R, [[R]_i, R', [R']_j], R_2)$$

The identity morphism for any state R in **States** is the tuple

$$id = (R, [], R)$$

consisting of an empty list.

A morphism in **States** will represent a behaviour of the system in question. According to the Basic State definition, behaviours are distinguished by the sequence of system states observed. This motivates the definition of a morphism $r : R \to R'$ in **States** as a tuple $(R, [R]_i, R')$, representing the states observed during the behaviour represented by r. However, not every sequence of states represents a behaviour which can be observed in a system. If a morphism $r = (R, [R]_i, R')$ is to represent a consistent behaviour, then there must be an ordering imposed upon this sequence so that it obeys the system constraints relating one state to the next. This is formalised by the following property.

Property 4.28 (The State Ordering Property). *A tuple $r = (R, [R]_i, R')$ satisfies the* State Ordering Property *if the following conditions hold:*

- *for all consecutive elements R_j and R_{j+1} in this tuple, there must exist some system models D and D' and valid state identification functors $\bar{R}_j$ and $\bar{R}_{j+1}$ such that*

$$R_j = D \circ \bar{R}_j \text{ and}$$
$$R_{j+1} = D' \circ \bar{R}_{j+1}$$

  and such that the following equalities hold for any component `fi` of the system :

  1. $\bar{R}_{j+1}(ficurrent) = \bar{R}_j(ficurrent)$ *or*
  2. $\bar{R}_{j+1}(ficurrent) = finext \circ \bar{R}_j(ficurrent)$

The State Ordering Property requires that for each transition observed in the course of a behaviour r, each component `fi` of the system must either stutter (1) or change state once (2). If `fi` changes state, (2) ensures that the constraints of `fi` relating one state to the next will be satisfied. This property is used to constrain the morphisms of Definition 4.27 so that they represent behaviours to which the system constraints can be applied.

Each of these behaviours, however, might not necessarily *obey* these system constraints. That is, the behaviour represented by a morphism r need not be consistent. If a morphism $r = (R, [R]_i, R')$ in **States** is to represent a consistent behaviour then the system models D used in the composition of each R_i in r must be sufficiently similar that the constraints relating one state to another are satisfied. An incorrect choice of system model can result in a failure to satisfy these constraints, and a subsequent inconsistency in the behaviour. We discuss this in more detail in the following section.

4.6.3 The Effect of System Models on the Category of States

Lemma 4.26 (the Consistency Lemma) summarises the conditions upon a functor $R : \mathbf{C}_V \to \mathbf{Set}$ in order for it to represent a consistent state. One of these

conditions is that there must exist some system model D and state identification functor $\bar{R}$ such that $R = D \circ \bar{R}$. When considering a morphism r of **States**, the choice of these system models for each R in r can affect whether this morphism represents a consistent behaviour. Specifically, for each pair R_i, R_{i+1} of consecutive objects in the tuple defined by r, there must exist some system model D such that both R_i and R_{i+1} are the result of composition with D and some state identification functors satisfying points (1) and (2) of Property 4.28. This ensures that the constraints in **C** relating one state to the next are satisfied by the states R_i and R_{i+1}.

This criterion can be formally modelled within the category **States** by identifying subcategories **States(D)**. Each such subcategory consists of those objects defined over a particular system model D.

Definition 4.29. *The category **States(D)** is comprised of*

- *those objects R of **States** for which there exists a valid $\bar{R} : \boldsymbol{C}_V \to \boldsymbol{C}$ such that*

$$R = D \circ \bar{R}$$

  for the system model D

- *those morphisms $r = \{R, [R]_i, R'\}$ of **States** for which every object R_i of $[R]_i$ is defined over some common D, where this composition satisfies the State Ordering Property.*

These morphisms are sometimes referred to as *consistent morphisms*. Consistent behaviours are precisely those represented by a morphism which is within **States(D)** for any system model D. Identifying these morphisms can be greatly simplified owing to the following theorem, which makes use of some of the characteristics of Basic State systems. Specifically, this theorem states that, to identify whether a morphism r is within some subcategory **States(D)**, it is only necessary to ensure that each pair of consecutive objects in r is within some common subcategory **States(D′)**. That is, it is not necessary to check that the Ds used for each pair are the same.

Theorem 4.30. *A morphism $r = (R, [R]_i, R')$ in **States** is within some subcategory **States(D)** if and only if each pair R_i, R_{i+1} within the tuple is the result of composing a common (to this pair only) system model D with state identification functors $\bar{R}_i, \bar{R}_{i+1}$ such that Property 4.28 is satisfied.*

Proof. Suppose the pair (R_i, R_{i+1}) is defined over system model D, while the pair (R_{i+1}, R_{i+2}) is defined over system model D', such that for both of these pairs Property 4.28 is satisfied. That is, each pair represents a consistent behaviour. In a Basic State system, the values of a variable in one state can be constrained relative to the values of a variable in the previous state only. This means there can be no system constraints preventing the triple (R_i, R_{i+1}, R_{i+2})

from representing a consistent behaviour, given that (R_i, R_{i+1}) and (R_{i+1}, R_{i+2}) both represent consistent behaviours. However, if (R_i, R_{i+1}, R_{i+2}) represents a consistent behaviour, then by Definition 4.29 this triple defines a a morphism within the subcategory **States(D″)** for some system model D''. By applying this reasoning to the entire morphism r, we see that r must be within a subcategory **States(D)** for some system model D. □

These states are then said to be pairwise consistent. This result means that we need only check each pair of states in a behaviour for membership of a common **States(D)** subcategory to ensure the entire morphism is within such a subcategory.

The generation of the category **States** has enabled us to formulate precise criteria for the consistency of a state. Furthermore, by examining the morphisms of this category we are able to identify characteristics of each system behaviour, as well as determine categorically if a behaviour is consistent. This category therefore provides a framework wherein we can analyse the two broad classes of inconsistency introduced in Section 4.5 within the wider context of system behaviours. In the following section, we examine the causes of a variety of types of inconsistency, and provide some suggested means of resolution. Throughout the discussion, we will make use of the category **States** to provide justification for the classifications of inconsistencies, and the suggested resolutions.

4.7 A Taxonomy of Inconsistencies

The category **States** provides us with a formal framework for discussing states and behaviours. By examining the characteristics of consistent and inconsistent states and behaviours within this category, we can create a taxonomy of inconsistencies. Each class of inconsistency will be characterised by common properties which are apparent when analysis is performed using **States**. We can then use these common properties to determine the effect of each of these inconsistencies upon the system as a whole, or to suggest methods of preventing or resolving the conflict. However, it is important to note that without detailed knowledge of the particular system in question, rating the severity or effect of an inconsistency can only be done using a very general metric.

The first type of inconsistency we discuss in this section is *singular component-wise inconsistency*, which is an example of failure of the component-wise consistency discussed in Section 4.5.1. Singular component-wise inconsistency occurs in specifications containing constraints which apply precisely after a certain number k state transitions. A conflict between these constraints results in an inconsistency which should affect only one single state — the state occurring after k state transitions. However, as we show in Section 4.7.1, the other states of the system will also demonstrate component-wise inconsistency, despite the

intuitive expectation that they should not be affected. To place this in perspective, Sections 4.7.2 and 4.7.3 introduce the concept of singular component-wise consistency. This is a weaker form of component-wise consistency which *is* demonstrated by these other states (assuming no additional inconsistencies arise). Section 4.7.4 describes how the behaviours of a system can also be affected by the presence of singular component-wise inconsistency, and suggests a means of resolving this.

Section 4.7.5 describes the type of inconsistencies which arise in a particular type of specification, known as a *behavioural* specification. We suggest a means of preventing this type of inconsistency, by borrowing from the Viewpoints [21] methodology. This illustrates how CCF can be used in conjunction with existing inconsistency management techniques. Section 4.7.6 provides a more in-depth discussion of the interaction-wise inconsistency of Section 4.5, while Section 4.7.7 describes a common inconsistency which affects behaviours rather than individual states. Together, the following sections form a taxonomy of inconsistencies which affect both states and behaviours. The inconsistencies discussed have been shown in work involving Rosetta to be the most commonly occurring types.

4.7.1 Constraints affecting a single state

It is possible to write specifications containing constraints which only apply to a single state, that state occurring after some k state transitions. A conflict between these constraints naturally means that the kth state of the component in question cannot be consistent. We have already seen these type of constraints in Basic State systems, where the value of variables in the initial state (that is, $k = 0$) is constrained relative to the data universe. For example, the following code defines a Basic State system, and will result in an inconsistent initial state of the component `f1`.

```
facet f1:: statebased
  x, y: int;
  begin
   T0:x@s0 = 0;
   T1:x@s0 = 1;
   T2:x' = 2;
   T3:y@s0 = 0;
end f1
```

Intuitively, it appears as though this conflict should affect *only* the initial state, as axiom $T2$ provides a value for x in every subsequent state. However, axioms $T0$ and $T1$ together have the effect of enforcing the equality $0 = 1$ within the system category $\mathbf{C}$. This equality will not be represented within the abstract state category $\mathbf{C}_V$, since these constraints originate in `f1` and not the data universe. This means that any state $R : \mathbf{C}_V \rightarrow \mathbf{Set}$ which factors through $\mathbf{C}$

will not preserve all countable coproducts of $\mathbf{C}_V$, as required by Lemma 4.26. That is, there is no state of this system which is both valid and component-wise consistent, despite the fact that the conflict between axioms $T0$ and $T1$ appears to affect the initial state only. In this section we will show how to more precisely estimate the effect of this inconsistency by defining a new type of consistency for a state, *singular component-wise consistency*. This will be a less strict version of component-wise consistency, holding in those states which satisfy only the axioms which pertain to this particular state.

Definition 4.31. *Given a system consisting of components* `f1, ... fn`*, a state* $R : \boldsymbol{C}_V \rightarrow \boldsymbol{Set}$ *of this system which occurs after each component has undergone* $k_1, ...k_n$ *state transitions respectively is* singular component-wise consistent *if*

- R *preserves finite limits and countable coproducts*
- *If* $(t1 = t2)$ *is a constraint in* `fi` *which constrains the values of variables of* `fi` *in the state occuring after* k_i *transitions of* `fi`*, and if* $v1$ *and* $v2$ *are morphisms in* $\boldsymbol{C}_V$ *representing these terms which are constrained, then* $R(v1) = R(v2)$

For example, if R represents any state *but* the initial state for the system specified by the code given above, then singular component-wise consistency requires that

- R preserves finite limits and countable coproducts
- $R(x) = R(2)$

Thus, although R cannot be component-wise consistent (since the cocone in $\mathbf{C}$ representing the integers is not the same as the cocone in $\mathbf{C}_V$ representing the integers), it is still singular component-wise consistent.

This type of inconsistency does not arise often in Basic State systems, as — with the exception of the initial state — these systems do not constrain individual states based upon the number of state transitions undergone. However, in the next chapter we will introduce Transition History systems, which permit constraints of the following forms

- `x@s0 = x@s10` (Relating states to each other using the number of transitions undertaken at any point)
- `x@s5 = 0` (Constraining the value of x after a particular number of state transitions)

These systems typically offer more opportunity for singular component-wise inconsistency to arise. In Section 4.7.4 we will show how identifying singular component-wise consistency of other states can be used to ascertain the severity of these conflicts. To do this formally, however, we must be able to interpret singular component-wise consistency in terms of CCF. The following section describes this.

4.7.2 A Formal Treatment of Singular Component-wise Consistency

In order to ascertain whether a state R observed after k transitions demonstrates singular component-wise consistency, we must construct a new category, denoted $\mathbf{C}^k$. This category will represent only those constraints which apply after precisely k state transitions. In fact, we should more accurately write it as $\mathbf{C}^{k_1,...k_n}$, where the system is composed of components `f1, ... fn`, and R is observed after these have undergone $k_1, ..., k_n$ transitions respectively (as in Definition 4.31). However, this complicates the notation and should instead be understood to be the case throughout the following discussion. This new category $\mathbf{C}^k$ will be known as the *singular consistency category* of R, where R is any state which is observed after these k transitions. By replacing the system category $\mathbf{C}$ with $\mathbf{C}^k$ in all discussions of component-wise consistency of R, we obtain discussions of singular component-wise consistency of R.

Constraints which will be represented as equalities in $\mathbf{C}^k$ are

- Constraints originating in the data universe.
- Constraints which relate values in the kth state to the data universe. For example, the constraint `rocket@s10 = LAUNCHED` should be included in the category $\mathbf{C}^{10}$, and will be used to determine singular component-wise consistency for any state R which might be observed 10 transitions into a countdown.
- Constraints which hold constantly, eg. `x=y`

Constraints which would *not* be within any singular consistency category $\mathbf{C}^k$ are those which constrain a value after k state transitions relative to a value in a different state. These include axioms of the form `x@s10 = x@s0` and `x' = x+1`.

We note that $\mathbf{C}^k$ does not include any information about how this 'k'-th state was reached, or whether indeed it can be reached consistently. That is, we are only interested in those constraints which must hold precisely after $k_1, ...k_n$ state transitions of each component, not those constraints which must have held earlier. To formally identify the constraints which should be represented in some particular $\mathbf{C}^k$, we first generate a category $\mathbf{C}(Free)$, which represents only those constraints originating in the data universe. That is, $\mathbf{C}(Free)$ represents only constraints which appear in *every* $\mathbf{C}^k$. One consequence of defining $\mathbf{C}(Free)$ in this way is that every valid state R factors through $\mathbf{C}(Free)$, regardless of the consistency of R or the number of state transitions that have been undergone by each component. That is, $\mathbf{C}(Free)$ simply contains the same information as the abstract state category $\mathbf{C}_V$, although it is represented differently.

For each possible singular consistency category $\mathbf{C}^k$, we then introduce a quotient functor I_k relating $\mathbf{C}(Free)$ and $\mathbf{C}^k$. If a valid state R occurring after k transitions factors through $\mathbf{C}^k$ by means of this quotient functor, R is said to demonstrate singular component-wise consistency. While $\mathbf{C}(Free)$ is not essential to defining singular component-wise consistency, it allows us to easily compare the criteria which different states must satisfy for singular component-wise consistency. With this in mind, we present the following definition.

Definition 4.32. *The category $\boldsymbol{C}(Free)$ is the category presented by the CCF sketch $(G, \mathcal{D}_V, \mathcal{L}, \mathcal{C})$ consisting of*

- *A graph G wherein nodes represent datatypes of the system and edges represent the variables, functions and constants*
- *A family $\mathcal{D}_V$ of pairs of paths in G, representing the system axioms constraining the data universe*
- *The family $\mathcal{L}$ of some of the cones of G*
- *A family $\mathcal{C}$ of some of the cocones of G*

The individual elements of this CCF sketch tuple have all been encountered before, as the following clarifications illustrate:

Clarification 4.33. *The graph G of Definition 4.32 is the graph G discussed in Clarification 4.4, and used in Definition 4.9 to generate the system category $\boldsymbol{C}$ of this system. Likewise, the cones and cocones $\mathcal{L}$ and $\mathcal{C}$ are the families of these names discussed in Constructs 4.6 and 4.7 and used in Definition 4.9 to generate the system category $\boldsymbol{C}$.*

The family $\mathcal{D}_V$, however, is not that used in Definition 4.9:

Clarification 4.34. *The family $\mathcal{D}_V$ consists of those pairs of paths discussed in Clarification 4.14, which represent constraints within the data universe only, and are used in Definition 4.16 to generate the abstract state category $\boldsymbol{C}_V$ of this system.*

Structurally, $\mathbf{C}(Free)$ resembles the system category $\mathbf{C}$, although the only constraints represented are those originating in the data universe. Since these are precisely the constraints represented in the abstract state category $\mathbf{C}_V$, every functor $R : \mathbf{C}_V \to \mathbf{Set}$ can be factored through $\mathbf{C}(Free)$. To describe this factorisation, we introduce the following functor.

Definition 4.35. *The* singular identification functor *is a functor $\bar{R}_k : \boldsymbol{C}_V \to \boldsymbol{C}(Free)$. It is* valid *if the following conditions are satisfied*

1. *$\bar{R}_k$ maps any object representing a datatype, or any morphism in $\boldsymbol{C}_V$ representing a function application, function or constant, to the object or morphism of $\boldsymbol{C}(Free)$ representing this same datatype, function aplication, function or constant.*

2. $\bar{R}_k(ficurrent) = finext^k \circ fi\text{-}init$
and for each component fi *and for any variable* x *of this component,*

$$\bar{R}_k(x_{fi}) = fi.x \circ finext^k \circ fi\text{-}init$$

where x_{fi} *and* $fi.x$ *are the morphisms representing the variable* x *of the component* fi *in categories* $\mathbf{C}_V$ *and* $\mathbf{C}(Free)$ *respectively.*

A singular identification functor $\bar{R}_k : \mathbf{C}_V \rightarrow \mathbf{C}(Free)$ is similar to the state identification functor $\bar{R} : \mathbf{C}_V \rightarrow \mathbf{C}$ of Definition 4.20. That is, $\bar{R}_k$ defines the number of state transitions k undergone by each component. Once again, $\bar{R}_k$ can instead be defined as a sketch morphism.

To formally define singular component-wise consistency, we now present the definition of the singular consistency category $\mathbf{C}^k$ for any k, and formalise the relationship between $\mathbf{C}(Free)$ and $\mathbf{C}^k$.

Definition 4.36. *For a system represented by the CCF sketch* $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$*, the* singular consistency category $\mathbf{C}^k$ *of any state* R *occurring after* k *transitions is the category presented by the CCF sketch* $(G, \mathcal{D}_k, \mathcal{L}, \mathcal{C})$ *where*

- G *is the directed graph of Clarification 4.4 used in Definition 4.9 to generate the system category* $\mathbf{C}$
- $\mathcal{D}_k$ *is a set of pairs of paths in* G *representing those axioms which apply precisely after* k *transitions, and those axioms constraining the data universe*
- $\mathcal{L}$ *and* $\mathcal{C}$ *are the cones and cocones of Constructs 4.6 and 4.7 used in Definition 4.9 to generate the system category* $\mathbf{C}$

The family $\mathcal{D}_k$ represents those system constraints which relate two terms which would both be visible in a state observed after k transitions. We show how to formally construct $\mathcal{D}_k$ in the following clarification.

Clarification 4.37. *The family* $\mathcal{D}_k$ *consists of those pairs of paths in* G *which define the data universe, and those pairs of paths* $((e_1, ...e_n), (e'_1, ...e'_m))$ *in* G *for which the following condition is satisfied.*

- *If the images of paths* $(e_1, ...e_n)$ *and* $(e'_1, ...e'_m)$ *in* $\mathbf{C}(Free)$ *are respectively* m_{Free} *and* m'_{Free}*, then there exist morphisms* m, m' *in the abstract state category* $\mathbf{C}_V$*, and a morphism* α *in* $\mathbf{C}(Free)$ *(which may vary with these morphisms* m, m'*) such that*

$$\bar{R}_k(m) = m_{Free} \circ \alpha \text{ and}$$
$$\bar{R}_k(m') = m'_{Free} \circ \alpha$$

for the valid singular identification functor $\bar{R}_k$ *indicating that* k *transitions have been undergone.*

The presence of the morphism α ensures that constraints which hold in every state, such as the constraint `x=y`, are represented in $\mathcal{D}_k$. Clarification 4.37 also formally identifies those morphisms in $\mathbf{C}(Free)$ which represent variables specifically in the k-th state (such as $m_{Free} = f1.x \circ f1next^k \circ f1init$, for a variable x of `f1`) simply by letting α be the identity. Using the singular consistency category of Definition 4.36 derived from this CCF sketch, we can define singular consistency of a state R.

Definition 4.38. *The k-th state $R : \boldsymbol{C}_V \rightarrow$ **Set** is singular component-wise consistent if there exists a valid singular identification functor $\bar{R}_k : \boldsymbol{C}_V \rightarrow \boldsymbol{C}(Free)$, and a functor $D_k : \boldsymbol{C}^k \rightarrow$ **Set** preserving finite limits and countable coproducts such that*

$$R = D_k \circ I_k \circ \bar{R}_{Free}$$

where I_k is the quotient functor $I_k : \boldsymbol{C}(Free) \rightarrow \boldsymbol{C}^k$

We note that in order to define singular component-wise consistency of a state R, we must select a point k at which it occurs. This is necessary in order to ensure that R satisfies all the constraints which hold at this k-th transition. It is possible for a state R to be singular component-wise consistent if it is observed as the k_i-th state but to lack this consistency if it is observed as the k_j-th state.

The interaction of these categories is shown in Figure 4.5, which will commute if R is singular component-wise consistent. We have included the interaction consistency category $\bar{\mathbf{C}}_V$, to show that a singular component-wise consistent state R may still be interaction-wise consistent. The system category $\mathbf{C}$ is also shown, so that this diagram may be compared to the commutativity conditions of Figure 4.4. Finally, we include several quotient functors denoted I, to show the relationships between some of these categories. We can use this formal framework to define the category of singular component-wise consistent states. This enables us to explicitly model the behaviours of a system displaying this particular type of inconsistency.

4.7.3 A Category of Singular Component-wise Consistent States

Objects in the category **States**, introduced in Section 4.6, are functors which satisfy all conditions of Lemma 4.26 (the Consistency Lemma). This means that **States** cannot be used to analyse systems in which some states will demonstrate singular component-wise inconsistency, since — as we discussed in Section 4.7.1 — there exist no consistent states of such systems. However, we can analyse these systems by using the category **Singular States**, which is the category of states which are both singular component-wise consistent and interaction-wise consistent.

Definition 4.39. *The category **Singular States** is comprised of*

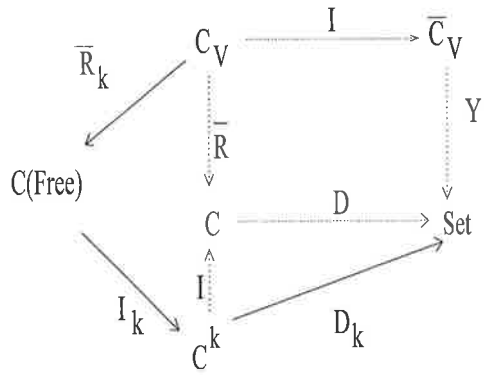

Figure 4.5: The interaction of categories we use to define singular component-wise consistency

- *objects which are those functors* $R : \boldsymbol{C_V} \rightarrow \boldsymbol{Set}$ *which are singular component-wise consistent (Definition 4.38) and interaction-wise consistent according to the conditions of Theorem 4.25*
- *morphisms which are tuples* $r = (R, [R]_i, R')$ *of* ***Singular State*** *objects satisfying the State Ordering Property (Property 4.28).*

Morphisms in the category **Singular States** will represent behaviours of the system. However, just as in the category **States**, not every morphism will represent a *consistent* behaviour, even allowing for singular component-wise inconsistency in the system. In the following section we discuss what level of consistency of behaviours can be expected in a system demonstrating singular component-wise inconsistency.

Morphisms in the category Singular States

In the category **States**, a morphism representing a consistent behaviour is a tuple of objects which satisfies the system constraints relating one state to the next. The category **Singular States**, however, allows for a less strict definition of consistency of behaviours. To reflect this, we will define a *singular consistent morphism* $r = (R, [R]_i, R')$ of **Singular States** to be one for which any system constraints relating two states in this tuple to each other are satisfied. That is, a singular consistent behaviour is one in which each state observed is singular component-wise consistent and the system constraints relating two of these observed states are also satisfied. However, system constraints relating an observed state to one which is *not* observed need not be satisfied in order for the behaviour to be singular consistent.

To identify whether a morphism r of **Singular States** represents a singular consistent behaviour, we generate the *singular consistent behaviour* category

$\mathbf{C}_r$. This category acts as the system category $\mathbf{C}$, but represents (as equalities) only those system constraints which must be satisfied if r is to be a singular consistent morphism of **Singular States**. One of the consequences of this is that every object R in the tuple r must factor through $\mathbf{C}_r$ if r is to represent a singular consistent behaviour. For any morphism r, the singular consistency category $\mathbf{C}_r$ is defined as follows.

Definition 4.40. *For a morphism $r = (R, [R]_i, R')$ of **Singular States**, $\boldsymbol{C}_r$ is the category presented by the CCF sketch tuple $(G, \mathcal{D}_r, \mathcal{L}, \mathcal{C})$ where*

- *G is the directed graph of Clarification 4.4, used to generate the system category $\boldsymbol{C}$.*
- *$\mathcal{D}_r$ consists of those pairs of paths of G with common source and target of the following form*
  - *paths within $\mathcal{D}_k$, as discussed in Clarification 4.37 and used in Definition 4.36 to generate the singular consistency category $\boldsymbol{C}^k$ for any state R of this tuple occurring after k state transitions (with the caveat that k is simply a shorthand for the several different numbers $k_1, ...k_n$ of state transitions undergone by each component).*
  - *paths relating a variable x to a variable in a state no more than i transitions from the current, where i is the length of the tuple r*
  - *paths relating a variable x in one state to a variable in another, where both these states are represented as objects in the tuple of r*
- *$\mathcal{L}$ and $\mathcal{C}$ are those families of cones and cocones of Constructs 4.6 and 4.7, used to generate the system category $\boldsymbol{C}$.*

There are therefore three types of pairs of paths in $\mathcal{D}_r$. The first type are those axioms which must be satisfied if each R of this tuple is to be singular component-wise consistent. The second type represent axioms of the form `x' = x+1` (relating future states to the current state), provided a sufficient number of state transitions is observed during r. The third type of pairs of paths in $\mathcal{D}_r$ represent axioms which explicitly relate two individual states, provided both these states are observed during r. These are axioms of the form `x@s0 = x@s10`, provided both `s0` and `s10` are observed as part of r. The following definition provides a formal criterion for r to represent a singular component-wise consistent behaviour.

Definition 4.41. *A morphism $r = (R, [R]_i, R')$ of **Singular States** will represent a singular consistent behaviour if there exists a functor $D_r : \boldsymbol{C}_r \to \boldsymbol{Set}$ preserving finite limits and countable coproducts, where for any object R_i in the tuple r*

$$R_i = D_r \circ I \circ \bar{R}_{Free}$$

where $\bar{R}_{Free} : \boldsymbol{C}_V \to \boldsymbol{C}(Free)$ is a valid singular identification functor according to Definition 4.35, and I is the quotient identity functor $I : \boldsymbol{C}(Free) \to \boldsymbol{C}_r$.

These are generally known as *singular consistent morphisms*. If every morphism r of **Singular States** is a singular consistent morphism, then singular component-wise consistency of a state implies component-wise consistency of this state. That is, there is no singular component-wise inconsistency within the system.

This category **Singular States** can be used to ascertain the extent of a conflict within the specification. We achieve this by examining states and behaviours to identify those which do not satisfy the necessary categorical conditions for singular consistency. The creation of **Singular States** makes this a more formal analysis than it otherwise might have been.

4.7.4 Determining and Resolving Component-wise Inconsistency

In a system where singular component-wise inconsistency is present, the severity of the conflict can be ascertained by the number of states which fail to demonstrate singular component-wise consistency. Similarly, the number of behaviours which are not singular consistent morphisms of **Singular States** can help us determine how severe the inconsistency is. However, it should be emphasised that both of these are very general metrics only and do not, for example, take into account the frequency of observation of any state, or the importance of any particular affected behaviour.

In general terms, it is not particularly useful merely to ascertain the effect of a conflict. Rather, we must also present some means of resolving or managing the inconsistency which results from the conflict. The technique we present here involves identifying and eliminating the constraints which cause the conflict. Specifically, we will use categorical techniques to generate a representation in CCF of a system which does not contain the conflicts leading to this inconsistency. From this categorical representation, we can then identify the changes which must be made to the specification itself. In general, we seek a new specification which is as similar to the original as possible, without containing the same inconsistency. However, as we discuss later, this goal is not always achievable or practical.

Resolving Component-wise Inconsistency

In this section we will present a technique for altering a system specification which demonstrates singular component-wise inconsistency. The altered specification will lack the conflicting constraints, and will therefore represent a consistent system. We will use categorical techniques to obtain this specification, identifying it by constructing its system category. The new specification should contain all constraints but those that conflict. That is, any singular consistent behaviour of the original system should also be a consistent behaviour of the

new specification. This will be the motivation behind the way we construct the system category representing the new specification.

Identifying the singular consistent behaviours of the original category requires us to examine **Singular States**. In Definition 4.41 we described the singular consistent morphisms r_i of **Singular States**, noting that their identification requires the construction of the singular consistent behaviour categories $\mathbf{C}_{r_i}$. These singular consistent behaviour categories were introduced in Definition 4.40 and represent the constraints which apply to each behaviour r_i. For each singular consistent behaviour r_i, the category $\mathbf{C}_{r_i}$ is generated from the CCF sketch $(G, \mathcal{D}_{r_i}, \mathcal{L}, \mathcal{C})$.

To generate the system category of the new specification, we need simply combine these sketches to produce the CCF sketch $(G, \mathcal{D}_{r_1} \bigcup ... \bigcup \mathcal{D}_{r_n}, \mathcal{L}, \mathcal{C})$. The element $\mathcal{D}_{r_1} \bigcup ... \bigcup \mathcal{D}_{r_n}$ of this sketch represents all the constraints from the original specification which must be satisfied if the morphisms r_i are to describe singular consistent behaviours (in the original specification). This CCF sketch presents the system category of the new specification. We will denote this new system category $\mathbf{C}_{r_1,...r_n}$, to emphasise those behaviours which the new and old specification have in common. Obviously, however, $r_1, ...r_n$ will be consistent in the new system, rather than merely singular consistent. For the code in Section 4.7.1, for example, we can define a category $\mathbf{C}_{r_1,...r_n}$ in which the only equalities that are present and do not originate in the data universe are those given by $T2$ and $T3$. This is then a completely consistent system, for which the states — with the exception of the initial state — are identical to the states of the original specification.

In this most general technique, the composition of the category $\mathbf{C}_{r_1,...r_n}$ is dependent upon the singular consistent behaviours of the system. Constructing this category inevitably means removing some axioms from the system (that is, $\mathbf{C}_{r_1,...r_n}$ represents fewer system axioms than $\mathbf{C}$). The least disruptive method is to remove the fewest constraints possible, and thereby construct a new system category $\mathbf{C}_{r_1,...r_n}$, representing every singular consistent behaviour. This means that we can observe any singular consistent behaviour r_1 of the original system as a consistent behaviour in the new system. However, in some cases this will not be possible, since the constraints which must be satisfied by one singular consistent behaviour r_i can conflict with the constraints which must be satisfied by another singular consistent behaviour r_j. In this case, the new specification will also suffer from singular component-wise inconsistency. Even when the constraints for different singular consistent behaviours do not conflict, it may be too time-consuming to identify each one of these behaviours in order to create the new specification.

Instead, a designer might choose to identify certain important or *critical* singular consistent behaviours. The criteria for a critical behaviour varies with the

system in question, and can be affected by such diverse factors as the primary function of the system, the proposed behaviour in the event of a safety concern, or even any refinements which are likely to be implemented. For example, a behaviour which the system may reasonably be expected to demonstrate often may be deemed more important than a behaviour which is observed only rarely. Equally, some behaviours may be rare, but when they do occur it is vital that they are performed correctly. In this way, we can see that the choice of critical behaviours is up to the designer to make, in consultation with the client. In this case, the system category representing the altered specification is $\mathbf{C}_{r_1,...r_i}$ where $r_1, ...r_i$ are these critical behaviours. If this specification also demonstrates singular component-wise inconsistency then this is an indication that the original specification requires further refinement. That is, the behaviours which were deemed critical are causing the inconsistency.

Designer Input and Singular Inconsistencies

The technique we suggested above for resolving singular component-wise inconsistency consists of eliminating system axioms until the conflict is located. Ideally we should eliminate as few axioms as possible, but in some cases it may be more practical to identify critical behaviours and eliminate the axioms which prevent these behaviours from being consistent. The task of identifying these critical behaviours must be performed by the designer. By presenting a resolution technique which allows for this user input, we recognise that certain behaviours and states are considered to be more important than others. We return to this concept of a critical behaviour in Chapter 6, wherein we discuss provisions to ensure that the ability to perform these behaviours is not affected by external circumstances such as power failure.

Another reason for allowing the designer to have input into the resolution process is to identify those inconsistencies which will be shortlived. An inconsistency which can be reasonably expected to be of short duration may be best addressed by simply ignoring it for the brief period in which it is present in the system. One example of this is an inconsistency which arises when a particular value of input is provided. This inconsistency may be resolved at any point by the provision of appropriate input. This type of inconsistency can then be said to have a higher chance of resolving itself than an inconsistency which is caused by variables which will not change their values. For example, the inconsistency in the following trivial specification has the potential to affect every state, but only if 'incorrect' input is provided.

```
facet f1(x: input int):: state-based
  private y: int;
  begin
   T0: IF x=0 THEN y=0 AND y=1 ELSE y=0;
   ...
end f1
```

However, if input of any value *other* than 0 is provided, the potential inconsistency has no effect. If a designer has independent knowledge of the system — such as the ability to conclude that an input value of 0 will never be provided — then he may choose to allow this inconsistency to remain. This extended reasoning about the likelihood and life of inconsistencies relies upon the provision of some details about the environment. That is, it relies upon input from the designer.

Permanent Inconsistencies

It is also worth noting that some component-wise inconsistencies can affect every state of the system. These are the inconsistencies which arise when a designer has accidentally included axioms which constrain the data universe. They hold in every state and are exemplified by the following constraint:

```
TO: x=5 AND x=6;
```

They have a particularly serious effect on the consistency of the system. There is no state R of a system containing this axiom which can be both valid and singular component-wise consistent. This implies that any alteration to the specification to remove this inconsistency is likely to involve a major refinement or re-examination of the requirements.

4.7.5 Inconsistencies In Behavioural Specifications

One existing framework for classifying and resolving inconsistencies is the Viewpoints [21] methodology. The specifications which use this method of inconsistency management tend to be component-based *behavioural* specifications. These are specifications in which each partition (module, component) describes a different behaviour of a single physical system. For example, in a behavioural specification of a telephone system, one module specifies the behaviour of the telephone when receiving a call while another specifies the behaviour of this same telephone when making a call. This is in contrast to the majority of specifications written in Rosetta, where each module describes the behaviour of a new physical component within the system.

The most common type of inconsistency in systems which are specified behaviourally occurs when two such specifications are combined, and is defined as follows.

Definition 4.42. Behavioural inconsistency *is the type of inconsistency which occurs when combining two behavioural specifications, where the behaviour described by one is disallowed by the other.*

In this case, the two components cannot be combined to describe the behaviours of a single system. While this inconsistency has been analysed with the Viewpoints methodology, we must characterise it using CCF in order to describe it within our taxonomy of inconsistencies. We do so by means of the following example.

The following code provides an example of a behavioural specification, being a partial specification of the audible responses obtained from a telephone system. Such a specification may then be used to describe a TDD system for the hearing-impaired. These are systems which translate the different types of noise on the phoneline to text, which is then displayed on a screen. If the noise on the phoneline is detected as a voice, a separate voice recognition program will translate the words to text.

The following partial component specification describes the behaviour of a telephone when receiving a call.

```
facet receiveCall:: statebased
  private Rcurrent-noise: [ready-dial-tone, ringing-tone, voice,
                  engaged-tone, no-tone];
  begin
   T0: Rcurrent-noise@s0 = no-tone;
   T1: IF (Rcurrent-noise=no-tone) THEN
          Rcurrent-noise' = ringing-tone;
   ...
end receiveCall
```

The following partial component specification describes the behaviour of a telephone when making a call.

```
facet makeCall:: statebased
  private Mcurrent-noise: [end-dial-tone, ringing-tone, voice,
                  engaged-tone, no-tone];
   begin
    L0: Mcurrent-noise@s0 = no-tone;
    L1: IF (Mcurrent-noise=no-tone) THEN
           Mcurrent-noise' = dial-tone;
    ...
end makeCall
```

Figure 4.6 depicts the complete statespaces of both these components, although for space considerations the complete code is not included. To describe all the possible behaviours and states of a single telephone, we must combine these two specifications. In the following section we describe how to identify those specifications which will give rise to behavioural inconsistency when combined.

Avoiding Behavioural Inconsistencies

The Viewpoints [76] technique provides certain conditions which, if satisfied, ensure that behavioural inconsistencies do not arise when combining behavioural specifications. These conditions are known as *methods*, and can vary with the system. However, there are some fundamental methods which apply in every

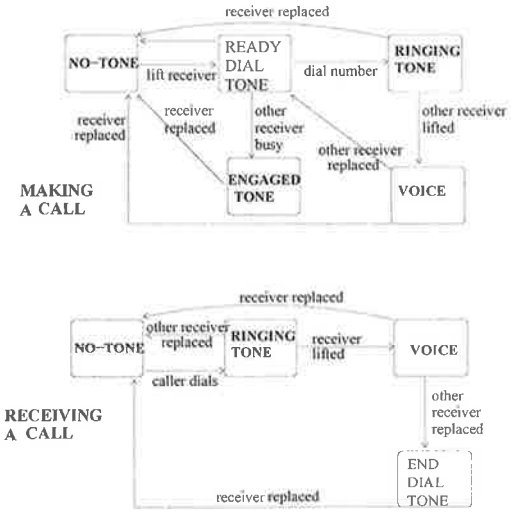

Figure 4.6: State-space diagrams for specification of an audible phone system

case and describe basic properties that two specifications must satisfy in order to be combined without behavioural inconsistencies. Here we paraphrase one such method, which expresses a relationship between the behaviours described by the specifications which are to be combined. In the language of this method, a state or state transition is *described by* a specification if it appears in the state space diagram of a component satisfying this specification.

Property 4.43. ***The Consistency Rule:***
Suppose a transition between two states $s1$ *and* $s2$ *is described by one component specification, and both* $s1, s2$ *are described by a second component specification. Then the second component must also describe a transition between* $s1$ *and* $s2$ *if the two specifications are to be combined to describe a consistent system.*

This translates into the vocabulary of CCF as follows:
Let $\mathbf{States}_{f1}$ and $\mathbf{States}_{f2}$ be the categories of states of component `f1` and `f2` respectively. Furthermore, suppose that both these categories contain objects R and R' representing two states of the system in question. The Consistency Rule then states that the presence of a morphism $r : R \to R'$ in $\mathbf{States}_{f1}(\mathbf{D})$ for some system model D means there must be some morphism $r : R \to R'$ in the category $\mathbf{States}_{f2}(\mathbf{D})$.

This condition must be satisfied if we are to combine two behavioural specifications to obtain a system which does not demonstrate behavioural inconsistency. The `makeCall` and `receiveCall` specifications of the telephone system

example satisfy this Consistency Rule, as can be deduced from their statespaces depicted in Figure 4.6. This means that we can combine these two specifications to obtain a complete specification of the behaviours and states of a single telephone.

The Viewpoints methodology also supports further consistency analysis, intended to be used when behavioural specifications describe separate instances of the component in question. In this case, the combination of these specifications describes what happens when two such components interact. For example, we can also describe two communicating telephones by combining `makeCall` and `receiveCall`. First, however, we must ensure that the behaviours specified by each component are compatible with those specified by the other. Briefly, this type of consistency analysis requires constructing a relationship between the states and behaviours of the two specifications, so that these synchronise. Once this relationship has been defined, further *methods* (or consistency rules) are used to ensure that the behaviours described by each specification are compatible with the relationship. Just as we did for the Consistency Rule, above, these methods can be translated into the vocabulary of CCF, thereby enabling us to create behavioural specifications which are guaranteed to be free of inconsistency. Because behavioural inconsistencies have been so thoroughly studied using other frameworks, we do not propose new metrics in CCF for estimating the severity of these. Instead, we simply present the above criteria for detecting and avoiding these conflicts.

4.7.6 Interaction-wise Consistency

Unlike the presence of component-wise inconsistencies, an interaction-wise inconsistency in a state R does not necessarily signal an error in the specification. This is because a state R which is interaction-wise inconsistent represents a combination of individual component states which cannot coexist. That is, the occurrence of an interaction-wise inconsistent state indicates that past system behaviour has resulted in an inconsistency. However, this does not imply that *every* behaviour will do so. The existence of an interaction-wise inconsistent state during a trace can indicate the serious problem of deadlock of a component or system. Deadlock occurs when no component can perform an action until another component does, if the system is to remain consistent. That is, no matter which component undergoes a state transition, the resulting system state will be interaction-wise inconsistent.

We can assign a rating to each behaviour $r = \{R, [R]_i, R'\}$ according to how badly it is affected by interaction-wise inconsistency. The simplest metric for these ratings is based upon the number of states observed throughout r which are interaction-wise inconsistent. However, as discussed in Section 4.7.4, a metric based purely on the number of affected states or behaviours is often too simplistic. In the case of interaction-wise inconsistencies, we must also consider

which parts of the system are affected by the inconsistency. Some components or subsystems may be deemed critical, while the behaviours of others are considered peripheral to the operation of the system. An inconsistency affecting a critical component is then more damaging to the system than an inconsistency affecting a peripheral component. Again, identifying which components are critical is a task undertaken by the designer, in consultation with the client.

This concept is explored further in Chapter 6, where we present a messaging subsystem which performs critical operations in a wider security system. We also discuss the possibility of letting inconsistencies remain, if they do not affect any of the critical systems or critical behaviours.

4.7.7 Sequential Behavioural Inconsistency

The final type of inconsistency we examine is related to the behaviours observed, rather than the individual states of a system. *Sequential behavioural inconsistency* occurs when a sequence of states, all of which are consistent and ordered correctly, still fails to represent a consistent behaviour.

Definition 4.44. *A morphism* $r = (R, [R]_i, R')$ *in* ***States*** *demonstrates* sequential behavioural inconsistency *when there is no system model* D *such that* r *is a morphism within* ***States(D)****.*

That is, in order to observe this behaviour we must allow multiple system models D to be used. Such a morphism r then describes an inconsistent behaviour. When examining such a behaviour, we can conclude that some state R_i is inconsistent, but the particular state R_i in question is not specified. Because of this, when examining the states observed during r, we might always conclude that the inconsistency is present in a state which is not the one currently under examination.

Because the inconsistency can always be presumed to be present in a different state, it is possible that many simple consistency checks will not detect sequential behavioural inconsistency. As a result, behaviours demonstrating this can often be accepted as "close enough" to consistent, and the inconsistency tolerated. However, it is advised that an indication should be provided to the user that sequential behavioural inconsistency has been found. The following section discusses how this indication might be provided, and what actions the user can take.

Resolving Sequential Behavioural Inconsistency

One way to hide the visible effects of sequential behavioural inconsistency is by using an interface. This interface may be triggered whenever sequential behavioural inconsistency is detected, thus alerting the user that these conflicts are present within the system. The interface chosen must be one which hides the affected variables. Specifically, if a variable is hidden under an interface,

then this variable may no longer be used to distinguish states. This means that there may be two system models D, D' which act differently upon the variables hidden by the interface, and yet identically on the visible variables. If the interface were not in effect then two states $D \circ \bar{R}$ and $D' \circ \bar{R}'$ would be visibly different. However, in the presence of the interface these appear as identical states, and so we may 'swap' system models D, D' without this being apparent. However, this relies on hiding every variable which is affected by the swap, in any state. We develop this idea further in Chapter 5, when discussing systems which communicate via input/output variables.

Identifying which variables should be hidden under the interface requires input from the user. This request for input can also be used to alert the user to the presence of sequential behavioural inconsistency. Specifically, the variables hidden should be those which are affected by this inconsistency. This technique, then, is only useful where the majority of variables are not affected. If most of the information in a system is affected by this inconsistency, then using an interface simply equates to hiding all this information, thereby drastically reducing the system's functionality. In addition, this technique should only be used when the designer determines that the effect of the inconsistency is reduced when the immediate visible effects are hidden. That is, this technique does not resolve the inconsistency, but simply allows a user to ignore it. Prompting the user for the abstraction mechanism or interface which will be used indicates to him that sequential behavioural inconsistency has been detected. In addition, this prompting allows the user to specify which effects of the inconsistency are to be hidden. In the event of serious repercussions to the consistency of the system, the user may choose not to specify an abstraction mechanism and instead attempt to resolve the inconsistency.

4.8 Inconsistency Analysis in CCF

Within the previous section, we have produced a taxonomy of inconsistencies as interpreted within the framework offered by CCF. As well as demonstrating how the different types of inconsistency appear within CCF, we have discussed ways to resolve or avoid each type. The classes of inconsistency we have discussed have included inconsistencies which affect individual states (such as component-wise or interaction-wise inconsistency), as well as those which affect behaviours (such as sequential behavioural inconsistency).

As well as discussing the immediate effect of each inconsistency, we have considered the effect that the conflict has on the system as a whole. For example, component-wise consistency is a property of an individual state. However, if conflicts within a specification prevent a state from displaying this type of consistency, then the system as a whole demonstrates *singular component-wise inconsistency*. This means that, to a lesser degree, the other states and behaviours of the system are affected by this conflict. Examining the degree to

which other states and behaviours are affected can therefore provide us with a metric to determine the severity of the inconsistency. We have also provided techniques by which the consistency of other states may be guaranteed, generally by removing the constraints which originally caused the conflict. By performing this within the categories of CCF, we are able to precisely judge the similarities between the original specification and the one resulting from these alterations.

We have also suggested less disruptive techniques for resolving an inconsistency which has a limited effect on the system as a whole. For example, in a behavioural specification the best way to resolve inconsistencies is to avoid their occurrance. To this end, we define some conditions which must hold if two behavioural specifications are to be combined to produce a consistent system. By contrast, in a system displaying sequential behavioural inconsistency, we simply hide the immediate visible effects. In cases where the choice of resolution technique depends upon the projected severity of the effect of the inconsistency, we have found it useful to define *critical* subsystems and behaviours. These are those components and transitions which should be affected only minimally, if at all, by any inconsistency. We return to this topic in the subsequent chapters, and demonstrate how these subsystems may be formally defined.

In summary, the category theoretical framework of CCF introduced in Section 4.1 has enabled us to formalise the relationships between components within a system. This formality means that we can identify inconsistencies by determining their underlying causes, resulting in the taxonomy of Section 4.7. One of the conclusions which can be drawn from this taxonomy is that particular types of specification are vulnerable to different errors, and susceptible to a wide variety of solution techniques. For example, if a system receives input, then an error involving one of these input variables might best be resolved by simply delaying any action until new input is provided. By contrast, an error involving the data universe (such as singular component-wise inconsistency) must be resolved by eliminating axioms until the conflict is resolved.

In the following chapter, we discuss how the different definitions of state and behaviour amongst systems can cause different types of inconsistency. This requires specifying a variety of systems using CCF, in order to compare the inconsistencies using the methods introduced in Section 4.7. As different systems present new inconsistencies, the methods of resolution suggested here can be customized to address the particular properties of each system. We also discuss the reasons for systems using different notions of state and behaviour, illustrating these with some real-world problems.

Chapter 5

Problem Solving Using CCF

5.1 Different Specifications and Problems

In the previous chapter we have shown how the categories of CCF can be used to identify and classify inconsistencies. So far, we have restricted our attention to Basic State systems, as defined in Section 4.2.1. This has made the generation of CCF less complex. However, the Basic State definition only defines one notion of 'state', and only permits one style of analysis. Systems which do not possess the characteristics of a Basic State system can contain inconsistencies which appear in different forms to those of a Basic State system. For example, singular component-wise inconsistency can occur when values in one state are related to values in an arbitrary state. Because Basic State systems constrain values in one state relative to values in the preceding state only, this type of inconsistency has few opportunities to arise in Basic State systems. Studying a variety of systems therefore provides information about what types of inconsistency might be expected in each system.

Another reason for studying a diverse collection of systems is because a specification may be analysed from many different perspectives during the design process. For example, a designer ensuring that a component satisfies safety regulations will be concerned with identifying those variables which take values outside a given safety range. On the other hand, a designer assessing the effect of removing a superfluous component will be concerned with the degree of interconnection between this component and the wider system. That is, the first designer is interested in values of variables, while the second designer is interested in the communication between components. By allowing designers to analyse systems from different perspectives, we will be able to use CCF to solve a variety of real-world problems. We will also be able to adapt existing analysis techniques (for example, from category theory or database analysis) and apply these to system specification problems.

In this chapter we will show how CCF can be used to describe systems other than Basic State systems, and to analyse these systems. We do this by first showing how the categories of CCF can be adjusted to capture the essential characteristics of each system. The two new classes of systems we will introduce are *Transition History* systems and *Input State* systems. In the former type of system, the number of state transitions undergone at any point is considered significant, and behaviours are distinguished based upon the interconnection of components. By contrast, *Input State* systems are simply Basic State systems augmented with input variables. We consider them here in order to demonstrate how the Basic State systems of Chapter 4 can be customized. We will then demonstrate some practical applications of each system type. Finally, for both types of systems we show how existing analysis techniques can be adapted to CCF and used with each system definition to solve common problems.

5.1.1 Overview of Specification Types

In Section 5.2 we present a characterisation of state and behaviour, which we term the *Transition History State* definition. Systems of this type view the number of state transitions undergone by any component as being sufficient to distinguish the state of the system. We also describe the way in which a Transition History State system characterises behaviour; namely, by emphasising the connection between components which must change state together for consistency reasons. Section 5.2.2 motivates this definition by discussing some of the applications where it is appropriate to use a Transition History State system. Sections 5.2.3 and 5.2.4 then show how CCF can be used to formally model a Transition History State system. This involves generating the system category and abstract state category to represent such a system and its states. Section 5.2.5 develops the conditions necessary for states of Transition History State systems to be consistent, while Section 5.2.6 shows how a category of these consistent states is generated.

In Section 5.2.7 we present a more in-depth example of a system appropriate for this type of specification, a Rubik's Cube specification. Many Rubik's Cube puzzles require the user to identify the shortest, or fastest, method of solving the cube. We briefly discuss some of the practical considerations of using computers to achieve this solution, as well as discussing how the shortest behaviour might be detected simply from observing a subsystem of the Cube. This provides the motivation for Section 5.3, in which we discuss how knowledge of the behaviour of a subsystem can provide information about the behaviour of the underlying system. In Section 5.3.1 we formalise the relationship between subsystem behaviours and system behaviours by introducing the concept of a minimal underlying behaviour. This is the shortest, or least disruptive, system behaviour which allows us to observe a particular subsystem behaviour. This section adapts some techniques of database analysis for use within CCF, thereby allowing us to deduce more about the behaviours of subsystems than would otherwise have been possible.

We then turn to a different class of systems, those in which input variables are used for communication. Section 5.4 discusses how this class of systems — known as *Input State* systems — is characterised. We then apply the same analysis to these systems as we did for Transition History State systems, showing how CCF can be used to represent Input State systems. One of the most common analysis questions, especially when considering communication between components, relates to identifying behaviours of these components which can co-exist. With this in mind, Section 5.5 introduces an existing analysis framework, **Span(Graph)**. This is an established framework which allows us to identify co-existing behaviours of components in a consistent system, such as that which may be specified using the Input State definition. We show how we can use CCF and the category of theories to identify co-existing behaviours in much the same way as **Span(Graph)** is used. We then show in Section 5.5.2

how the results obtained using CCF are equivalent to those obtained by using **Span(Graph)**.

5.2 The Transition History State Definition

In this section we will present a CCF analysis of a new class of systems, the *Transition History State* systems. These characterise states and behaviours as described below.

5.2.1 Characteristics of a Transition History State System

Definition 5.1. *A* Transition History State *(THS) system is one satisfying the following assumptions.*

1. *The number of state transitions undergone by each component at any point is sufficient to distinguish states of the system.*
2. *Values of variables in one state may be constrained relative to values of variables in any other state, based upon the number of state transitions undergone by each component.*
3. *System behaviours are distinguished by those components which change state together during each behaviour in order to preserve consistency.*

Similarly, a *Transition History state* is a state of such a system, and a *Transition History behaviour* is a behaviour of such a system. Definition 5.2 is intended to be an informal definition only, and in subsequent sections we will clarify what is meant by each of these points.

In Basic State systems the values in the next state can be constrained relative to values in the current state only. However, languages such as Rosetta permit axioms which relate values in one state to values in another, arbitrary, state. Such axioms include those of the form `x'' = x+1`, which relates the current value of a variable `x` to its value after two state transitions, or `x@s5 = 0`, which restricts the values which can be observed after five state transitions. Unlike Basic State systems, THS systems allow these types of constraints within a specification. One consequence of these constraints is that in order to predict the next state, it is no longer sufficient to know the current values of all variables (even in the absence of non-determinism). This is demonstrated by the following THS specification, which gives rise to the partial state-space diagram shown in Figure 5.1.

```
facet f1:: state-based
  private x, y: int;
  begin
   T0: x@s0 = y@s0 = 0;
   T1: x'' = x;
```

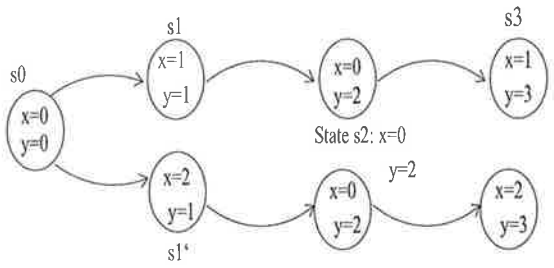

Figure 5.1: Knowing the current values of variables is not sufficient to predict the future values of a THS system

```
   T2: y' = y+1;
end f1
```

State `s2` in this figure refers to the (Basic) state wherein `x = 0, y = 2`. When this system is in state `s2`, in order to predict the values of variables in the next state we must know the values of variables in the previous state (ie. `s1` or `s1'`). By contrast, in a Basic State system the states following `s2` cannot be determined by the states which preceded `s2`.

Another way in which THS systems differ from Basic State systems is by the importance given to the number of state transitions undergone by each component. In a THS system, the number of state-changes encountered at any point through a trace distinguishes the current state of the system (although stuttering is still treated as being irrelevant). It is possible to extend Basic State specifications to describe THS systems, simply by adding a 'counter' variable tracking state changes to each component. However, Rosetta was originally intended to describe THS systems only. Because of the historical importance of these THS systems to Rosetta and therefore to this thesis, we will extend the categories of CCF to provide a way to model THS systems without simply interpreting them as an extension of Basic State systems.

THS systems also characterise behaviours differently to Basic State systems. The Basic State definition distinguishes behaviours by the states observed. This method lends itself well to identifying and resolving inconsistencies, which are associated with the values of variables at any point in time. However, there are other forms of analysis performed upon a system specification which require a different characterisation of behaviours. For example, if a designer wishes to assess the impact of removing a component, he will be less concerned with the values of variables than with how this removal affects the wider system. For this type of analysis, it is therefore more useful to distinguish behaviours of a system based upon the inter-connection of components. That is, if a component communicates with others (such as compelling them to change state via the

use of global variables), then removing this component will have an impact on the other components. To examine the severity of this impact we need a characterisation of system behaviour which identifies those components which must act in lockstep at any given time. We will provide this characterisation for THS systems.

5.2.2 Applications of THS Systems

The preceding discussion of the differences between Basic State systems and THS systems provides an indication of where the latter type of specification might be used. In general, a THS specification is appropriate for a system exhibiting finite, or terminal, behaviours. These include mechanical systems which undertake a finite number of transitions in order to achieve a given goal. The specification of "George" [1], a controller for a boat, is an example of this. Another example of a system which might be modelled by a THS specification is a Rubik's Cube puzzle. We will explore in more detail in Section 5.2.7, illustrating how each coordinate on a facet of the Cube can be defined to be represented by a separate component of the system. The state of each component is then dependent upon the colour at this coordinate. A state transition in such a system is a particular rotation of the Cube. Because rotations require a number of components to change state simultaneously, they are appropriately modelled by a Transition History behaviour, as we discuss in Section 5.2.7.

5.2.3 The CCF Sketch Representing a THS System

In Chapter 4, we introduced techniques for analysing Basic State systems using CCF. These were based on the notion of CCF sketches, which we used to generate categories. These categories then formed the structure of CCF, as used to analyse Basic State systems. We use the same technique to analyse THS systems, thereby preserving the homogeneity of our categorical approach and demonstrating the flexibility of CCF.

Definition 5.2. *The* system category *for a THS system is the category* $\mathbf{C}$ *presented by the CCF sketch* $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ *where*

- G *is the directed graph of Clarification 4.4, used in Definition 4.9 to generate the system category of Basic State systems*
- $\mathcal{D}$ *is the family of pairs of paths in* G *discussed in Clarification 4.5, and also used in Definition 4.9*
- $\mathcal{L}$ *is the family of cones in* G *constructed according to Construct 4.6, and also used in Definition 4.9*
- $\mathcal{C}$ *is the family of cocones in* G *constructed according to Construct 4.7, and also used in Definition 4.9*

Although the CCF sketch representing a THS system is constructed identically to the CCF sketch representing a Basic State system, we note that the constraints (represented by $\mathcal{D}$) of a THS system may be different to those of a Basic State system, as detailed in Definition 5.1.

As for a Basic State system, a system model of a THS system is a functor $D : \mathbf{C} \rightarrow \mathbf{Set}$ which preserves finite limits and countable coproducts.

5.2.4 The CCF sketch Representing a Transition History state

One of the characteristics of systems specified using the THS definition is that the number of transitions executed up to some point can distinguish the state of the system. We incorporate this dependency into the definition of the *abstract state category* for a THS system represented by the CCF sketch $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$. The abstract state category will identify those elements of the system which are visible in a single state.

Definition 5.3. *The* abstract state category $\boldsymbol{C_V}$ *of a THS system is the category presented by the CCF sketch* $(G_V, \mathcal{D}_V, \mathcal{L}, \mathcal{C})$ *where*

- G_V *is a directed graph wherein nodes represent datatypes of the system and edges represent functions, constants and single values of variables in this THS system*
- $\mathcal{D}_V$ *is a family of pairs of paths in* G_V *with common source and target, representing the axioms within the data universe of the specification*
- $\mathcal{L}$ *is a family of some of the cones in* G_V
- $\mathcal{C}$ *is a family of some of the cocones in* G_V

We have encountered all these elements of the tuple in different contexts before this:

Clarification 5.4. *The graph* G_V *of Definition 5.3 is comprised of*

- *Nodes which are identical to the nodes of the graph* G*, used in Definition 5.2 to generate the system category of this THS system.*
- *Edges which represent* all *functions (including the* `next`*,* `init` *and* `current` *functions), constants and variables of each component , where variables are implemented as functions with a null domain.*

The graph G_V for a THS system differs from the graph G_V for a Basic State system (detailed in Clarification 4.13) only by including the morphisms `fi-init` and `finext` for any component `fi`. These functions can be used to define the `fi-state` datatype, as discussed in Appendix A. Their inclusion in the graph G_V for a THS state allows us to impose an ordering upon the

`fi-state` datatype in THS systems (for any component `fi`). This ordering will enable us to distinguish between two states which differ only in assigning different values to the `ficurrent` variable for any `fi`. That is, we will be able to distinguish two states which differ only in the number of state transitions undergone by each component. We discuss this further later.

Clarification 5.5. *The family $\mathcal{D}_V$ consists of some of those pairs of paths in G_V with common source and target. These paths represent those pairs of functions or constants which are within the data universe and constrained to be equal by data universe axioms.*

The family $\mathcal{D}_V$ is identical to the family $\mathcal{D}_V$ of paths detailed in Clarification 4.14 and used in Definition 4.16 to generate the abstract state category of a Basic State system.

Construct 5.6. *The families $\mathcal{L}$ and $\mathcal{C}$ are identical to the families of cones and cocones used in Definition 5.2 to generate the system category $\mathbf{C}$.*

These families represent product datatypes and constructively specified datatypes respectively. We emphasise the following consequence of this definition of $\mathcal{C}$.

Remark 5.7. *For any component `fi` within a THS system, the `fi-init` and `finext` morphisms describing the `fi-state` datatype are represented by cocone injections and universal morphisms in the abstract state category. That is, we use these functions to constructively specify the `fi-state` datatype, and accordingly include a cocone denoting this datatype in $\mathcal{C}$ when generating $\mathbf{C}_V$.*

The CCF sketch we describe here differs from the CCF sketch of Definition 4.12 (which presents the abstract state category for a Basic State system) only by including a representation of the `fi-init` and `finext` functions for each component `fi`, and by representing the `state` datatype as a coproduct of ones using these functions.

Figure 5.2 shows the abstract state category $\mathbf{C}_V$ of Definition 5.3 for the code of Section 4.3.4, interpreted as a THS system. As this is the same code which was used to illustrate CCF for a Basic State system, we may usefully compare the differences apparent between Figure 4.2 and Figure 5.2. As we did for Basic State systems, we define a state of a THS system to be a functor $R : \mathbf{C}_V \rightarrow \mathbf{Set}$.

5.2.5 Consistency of Transition History States

Given the system category $\mathbf{C}$ and abstract state category $\mathbf{C}_V$ of a THS system, we can formulate the categorical criteria for a state $R : \mathbf{C}_V \rightarrow \mathbf{Set}$ of a THS system to be consistent. These are the same essential criteria as those required for a Basic State to be consistent, and so we refer the reader to Section 4.5 for a complete discussion.

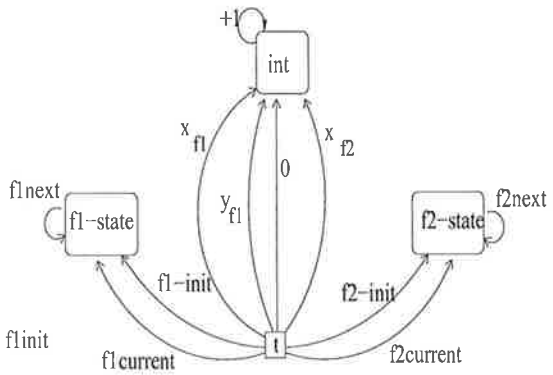

Figure 5.2: The abstract state category $\mathbf{C}_V$ of a THS system. The system category is the same whether this is interpreted as a Basic State system or a THS system, and is shown in Figure 4.1

Component-wise consistency of a Transition History state

Just as for a Basic state, a Transition History state must factor through the system category $\mathbf{C}$ if it is to be component-wise consistent. This requires the use of a state identification functor as introduced in Definition 4.20. This will be a functor $\bar{R} : \mathbf{C}_V \rightarrow \mathbf{C}$ from the abstract state category $\mathbf{C}_V$ of Definition 5.3 to the system category $\mathbf{C}$ of Definition 5.2. A valid state identification functor of a Transition History state is one which satisfies Definition 4.20, when it is applied to these categories of Definition 5.3 and 5.2. The different compositions of the abstract state category $\mathbf{C}_V$ for Basic State and THS systems mean that such a functor will have some additional characteristics in a THS system. This is because the morphisms `fi-init` and `finext` are not represented in the abstract state category of a Basic State system, although they are in the abstract state category of a THS system.

Remark 5.8. *For a THS system, a valid state identification functor $\bar{R}$: $\mathbf{C}_V$ $\rightarrow$ $\mathbf{C}$ according to Definition 4.20 takes the morphisms in $\mathbf{C}_V$ representing the `fi-init` and `finext` functions to the morphisms in $\mathbf{C}$ representing these functions.*

The necessary conditions for component-wise consistency of a THS state are the same as for a Basic state. These are given in the conditions of Theorem 4.21.

Interaction-wise consistency of a Transition History state

To be interaction-wise consistent, a Transition History state must factor through an interaction consistency category. Such a category will, as discussed in Section 4.5.2, represent constraints which require a shared variable to have the same value when viewed from any component. Definition 4.24 describes how

to construct the CCF sketch presenting the interaction consistency category, given the CCF sketch presenting the abstract state category. This technique is used for both THS and Basic State systems, notwithstanding the difference btween the CCF sketches presenting the abstract state category for a Basic State system and THS system. Briefly, it consists of augmenting the CCF sketch presenting the abstract state category with the constraints $\bar{\mathcal{D}}_V$ representing shared variables.

Definition 5.9. *The* interaction consistency *category $\bar{C}_V$ for a THS system is presented by the CCF sketch $(G_V, \mathcal{D}_V \bigcup \bar{\mathcal{D}}_V, \mathcal{L}, \mathcal{C})$, where*

- *G_V is the graph used in Definition 5.3 to generate the abstract state category of the THS system*
- *$\mathcal{D}_V$ is the family of pairs of paths used in Definition 5.3 to generate the abstract state category of the THS system*
- *$\bar{\mathcal{D}}_V$ is the family of pairs of paths in G_V detailed in Clarification 4.23, and representing constraints which ensure shared information has the same value when viewed from any component*
- *$\mathcal{L}$ and $\mathcal{C}$ are the families of cones and cocones of G_V used in Definition 5.3 to generate the abstract state category of the THS system*

The necessary conditions for interaction-wise consistency of a THS state are the same as for a Basic state. These are given in the conditions of Theorem 4.25.

A valid and consistent Transition History state

A valid and consistent Transition History state $R : \mathbf{C}_V \to \mathbf{Set}$ is one which satisfies the conditions of Lemma 4.26, the Consistency Lemma. This lemma requires that R preserve finite limits and countable coproducts, as well as satisfying the conditions for interaction-wise consistency and component-wise consistency. Given the construction of the abstract state category $\mathbf{C}_V$ for a THS system (formalised in Definition 5.3), we emphasise the following consequence of this lemma.

Remark 5.10. *A valid Transition History state $R : \boldsymbol{C}_V \to \boldsymbol{Set}$ must preserve the countable coproduct of ones in $\boldsymbol{C}_V$ representing the `fi-state` datatype for any component `fi`. This cocone is therefore mapped to a countable coproduct of ones in **Set**.*

That is, the `state` datatype for each component is modelled as a countable ordered set. Because the abstract state category $\mathbf{C}_V$ contains a representation of the functions `init` and `next` which are used in the specification to define an ordering upon states, we can deduce from Remark 5.8 the following conclusion:

Lemma 5.11 (Uniqueness Lemma). *For any valid and component-wise consistent Transition History state $R : \boldsymbol{C}_V \to \boldsymbol{Set}$ there is a* unique *valid state identification functor $\bar{R} : \boldsymbol{C}_V \to \boldsymbol{C}$ for which $R = D \circ \bar{R}$, for any system model D.*

Proof. From the criteria of Lemma 4.26, we know that the image under R of the coproduct of ones representing the `fi-state` datatype (for any component `fi`) in $\mathbf{C}_V$ will be a countable set, ordered by the image under R of the `fi-next` morphism. This means that there is some unique k for which

$$R(ficurrent) = R(finext^k \circ fi\text{-}init) \ (*)$$

The Consistency Lemma states that for any consistent state R, there is some valid state identification functor $\bar{R}$ for which $R = D \circ \bar{R}$ for some system model $D : \mathbf{C} \to \mathbf{Set}$. Every system model D preserves countable coproducts, and the `fi-state` datatype for any component `fi` is represented in $\mathbf{C}$ as a countable coproduct of ones. Using the fact that D therefore preserves the countable product of ones representing the `fi-state` datatype, along with Remark 5.10, (*) and the 'identity' action of $\bar{R}$ on functions represented in both $\mathbf{C}$ and $\mathbf{C}_V$ (such as `fi-init` and `finext`) detailed in Definition 4.20, we can deduce

$$\bar{R}(ficurrent) = \bar{R}(finext^k \circ fi\text{-}init)$$

for this same unique k. Together with the other points of Definition 4.20, this completely fixes the action of the state identification functor $\bar{R}$ in question, and therefore uniquely identifies it. □

5.2.6 The Category of Transition History States

In Section 4.6 we introduced **States**, the category of states of a Basic State system. In this section we will define the category **TH States**, which will be the category of states of a THS system. For any THS system, the **TH States** category is comprised of objects which are consistent Transition History states of this system, and morphisms representing its Transition History behaviours.

Definition 5.12. *The category **TH States** is comprised of*

- *objects which are functors $R : \mathbf{C}_V \to$ **Set** satisfying the conditions of Lemma 4.26*
- *morphisms which are tuples $(R, [M]_i, R')$ satisfying Construct 5.13 and Property 5.14 (defined below as the THS State Ordering Property).*

In this definition, $[M]_i$ is a list of the state transitions which are undergone during r, describing those components which are compelled to change state together during each transition. We discuss this in more detail below.

Objects in the category of Transition History States

Two objects R and R' will represent identical system states if and only if R and R' are naturally isomorphic functors. Lemma 5.11 states that for each object R in **TH States**, there is a unique valid state identification functor $\bar{R}$ such that $R = D \circ \bar{R}$ for some system model D. The action of this state identification functor upon the morphism $ficurrent$ of $\mathbf{C}_V$ for any component `fi` is sufficient

to distinguish two states. This is because, for a THS system, the functions `init` and `next` impose an ordering on the `state` datatype, and are represented in $\mathbf{C}_V$. That is, there can be no natural isomorphism between two functors R and R' for which

$$R(ficurrent) = R(finext^j \circ fi\text{-}init) \text{ while}$$
$$R'(ficurrent) = R'(finext^k \circ fi\text{-}init)$$

if $j \neq k$. This means that two otherwise identical states R and R' which are observed at different points during a behaviour will be represented by non-isomorphic objects in **TH States**.

Morphisms in the category of TH States

Morphisms in **TH States** represent behaviours of the THS system in question. Unlike behaviours of a Basic State system, Transition History behaviours are distinguished by the components which are compelled to change state together to preserve consistency. Morphisms in **TH States** must therefore distinguish between a system behaviour in which components change state in lockstep because they are compelled for reasons of consistency, and a system behaviour in which components change state simultaneously, but are not compelled to do this.

We have defined a morphism in **TH States** to be a tuple $r = (R, [M]_i, R')$ where $[M]_i$ is a list identifying the components which change state together during each state transition described by r. Each element M_i of the list $[M]_i$ therefore describes a single state transition occurring during r, and lists the components which change state during it.

Clarification 5.13. *A morphism $r = (R, [M]_i, R')$ of **TH States** is a tuple wherein each element M_i of the list $[M]_i$ is a family of pairs (fj, k) of the form `(component-name, transition number)`.*

Each pair identifies a component (eg. `fj`) which has undergone a certain number (eg. $k-1$) transitions. Furthermore, the components identified by the family of pairs in each M_i change state simultaneously (for reasons of consistency) at this point in the behaviour described by r. That is, this element M_i describes what occurs in the system when the component `fi` undergoes its `k`th transition, given the behaviour described by r up to this point.

As mentioned earlier, morphisms in **TH States** must distinguish between behaviours in which components are compelled to act in lockstep, and behaviours in which the components act in lockstep but are not compelled to do so. To achieve this, we require that each family M_i in $[M]_i$ must be *minimal*. Minimality of M_i means that this family must not contain a pair $(\texttt{fi}, q)$ where `fi` is not compelled to undergo its next state change simultaneously with the other components listed in M_i. In this case, we must create another element M_{i+1}

in the list, to represent this unrelated state transition $(\texttt{fi}, q)$. The relationship between any two components `fi` and `fj` listed in M_i may be the result of *transitive* connections. That is, `fi` need not share information with `fj`. Instead, the state transition of `fi` might compel another component, `fm`, to change state — which then compels `fj` to change state. In this case, all three components changing state (`fi`, `fm` and `fj`) must be listed in M_i.

By describing the number of state transitions undergone by any component (if a component is not listed in $[M]_i$ we assume it has not changed state), we can obtain from $[M]_i$ a list of state identification functors $\bar{R}_i$. As has been previously discussed, these state identification functors also describe the number of state transitions undergone by each component, and are therefore equivalent to the list $[M]_i$.

As is the case in **States**, not every tuple will represent a consistent behaviour. If a tuple $r = (R, [M]_i, R')$ is to represent a consistent behaviour, there must be some ordering imposed upon the state transitions represented by r. Specifically, we require that the ordering of the elements within $[M]_i$ is such that r represents a sequence of state changes which obey the ordering supplied by the `next` function. This is formalised by the following property.

Property 5.14 (The Transition History State Ordering Property). *A tuple* $r = \{R, [M]_i, R'\}$ *satisfies the* Transition History State Ordering Property *if the following conditions hold:*

1. *For any two elements* M_a *and* M_c *of* $[M]_i$ *where* M_a *contains the pair* $(\boldsymbol{fx}, j)$ *and* M_c *contains the pair* $(\boldsymbol{fx}, k)$*, if there is no element* M_b *of* $[M]_i$ *where* $a \leq b \leq c$ *which mentions the component* $\boldsymbol{fx}$*, then* $k = j + 1$*.*

2. *If an element* M_a *of* $[M]_i$ *contains a pair* $(\boldsymbol{fx}, j)$ *and is the last element in this list to mention the component* $\boldsymbol{fx}$*, then if* $\bar{R}'$ *is the unique state identification functor associated with* R' *the following equality holds*

$$\bar{R}'(fxcurrent) = fxnext^j \circ fx\text{-}init$$

   for this same j*.*

3. *If an element* M_a *of* $[M]_i$ *contains a pair* $(\boldsymbol{fx}, j)$ *and is the first element in this list to mention the component* $\boldsymbol{fx}$*, then if* $\bar{R}$ *is the unique state identification functor associated with* R *the following equality holds*

$$\bar{R}'(fxcurrent) = fxnext^j \circ fx\text{-}init$$

   for this same j*.*

4. *If there is no element of* $[M]_i$ *mentioning the component* $\boldsymbol{fx}$ *then the following equality holds*

$$\bar{R}'(fxcurrent) = \bar{R}(fxcurrent)$$

Condition 1 of Property 5.14 ensures that the number of state transitions of a component increments by one each time this component undergoes a state change described by r. Conditions 2 and 3 ensure that there can be no 'gaps' between the initial state and the first state transition described by r (likewise the final state and last state transition of r). Finally, Condition 4 ensures that every component which undergoes a state change during the behaviour described by r is mentioned in some element of $[M]_i$. These last 3 conditions rely on the Uniqueness Lemma to identify the unique state identification functor $\bar{R}$ associated with each TH state R. The morphisms $r = (R, [M]_i, R')$ of **TH States** are sometimes known as *valid* tuples.

Equality and composition of morphisms

A morphism $r = (R, [M]_i, R')$ of **TH States** might contain *independent* families M_j and M_k of $[M]_i$. Families M_j and M_k of $[M]_i$ are independent if there is no component `fi` mentioned within both families. Given the description of a Transition History behaviour in Definition 5.1, it is clear that the relative ordering of independent families should be irrelevant when it comes to distinguishing behaviours. As a consequence, equality of morphisms is defined up to the rearrangement of independent families, provided the THS Ordering Property is still satisfied.

Definition 5.15. *Two morphisms $r = (R, [M]_i, R')$ and $r' = (R, [M]_j, R')$ of **TH States** will be equal if for any element $M_a \in [M]_i$, there exists an identical element $M_a \in [M]_j$, and vice versa.*

Composition of morphisms may then be defined as simple concatenation of the relevant lists.

Definition 5.16. *Concatenation of two valid morphisms $r = (R, [M]_i, R')$ and $r' = (R', [M]_j, R'')$ is defined to be the morphism*

$$r' \circ r = (R, [[M]_i, [M]_j], R'').$$

Of course, given the definition of equality, this concatenation may also be represented as interleaving of the independent elements of each list $[M]_i, [M]_j$. However, for the sake of convenience we will generally use the notation above. We emphasise again that a morphism in **TH States** is not intended to describe what is visible throughout a behaviour, but rather to describe state transitions in which components are compelled to act simultaneously.

The Effect of System Models on the category TH States

When introducing the category **States**, we noted that the choice of system models D affects whether or not a morphism r of **States** represents a consistent behaviour. This is discussed more fully in Section 4.6.3, and is also an issue which must be considered in **TH States**.

In Section 4.6.3, we concluded that a morphism $r = (R, [R]_i, R')$ of **States** represents a consistent behaviour if every state in this tuple can be obtained by composition with a common system model D. This conclusion holds within **TH States** also, although the different nature of morphisms in the category requires that this property is phrased differently. To facilitate this, we define a subcategory **TH States(D)** of **TH States** for each system model D.

Definition 5.17. *The category **TH States(D)** is comprised of*

- *objects $R : \boldsymbol{C_V} \rightarrow$ **Set** of **TH States** which are formed by composition with D*
- *morphisms $r = (R, [M]_i, R')$ of **TH States(D)** where the list $[M]_i$ defines a sequence of state identification functors $[\bar{R}]_i$ such that for every such $\bar{R}_i$, the functor $D \circ \bar{R}_i$ is an object in **TH States(D)**.*

A morphism r of **TH States** which represents a consistent behaviour is one which is within some subcategory **TH States(D)**.

5.2.7 An Example THS System: The Rubik's Cube

One type of system which can be appropriately specified using the Transition History State definition is a Rubik's Cube. A Rubik's Cube is a solid (traditionally a cube, but [58] describes some variants) which is divided into a number of rotatable rows and columns. Puzzles involving the Rubik's Cube ask for the rows and columns to be rotated so that a particular pattern is obtained upon each face of the solid. The best-known of these puzzles is the one where the Cube must be 'solved', that is, put into a configuration in which each face is a different colour.

We will specify a Rubik's Cube as a system comprising a number of components. Each of these components represents a single coordinate upon the Cube. The system we use for identifying the components involves numbering the faces as in Figure 5.3. This identification scheme is presented in [58], along with some alternative ordering systems. The state of any component is determined by the colour displayed on the Cube at the associated coordinates, while a state transition of each component corresponds to a change in the colour of the associated coordinate. State transitions are prompted by a user performing a rotation on the Cube.

A Rubik's Cube Example

The following partial Rosetta specification defines some of the types and functions which would be used to specify a (cubic) Rubik's Cube.

```
domain CubeDefinitions:: statebased
 Rotation:[left column rotate 90; middle column rotate 90;...;
           top row rotate 90;...; bottom row rotate 90]
```

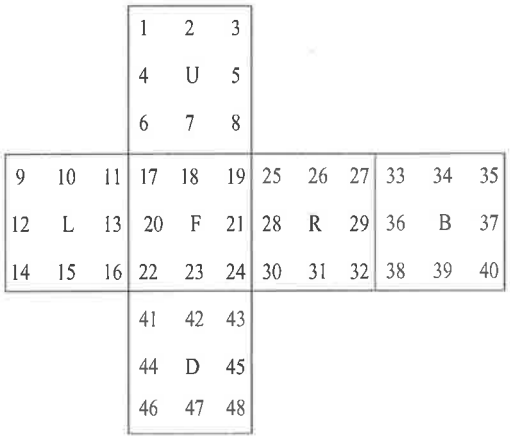

Figure 5.3: The reference system for squares on a Rubik's Cube

```
// Rotation is a type giving the possible rotations of the Cube
   a user can make (for space considerations we do not list
   all these rotations in this partial specification)

function getnextcolour: Rotation, colour -> colour
// This function describes the effect upon a coordinate of
   a given rotation. The function definition will be dependent
   upon the particular Cube
  ...

end CubeDefinitions
```

The `CubeDefinitions` domain describes the datatypes and functions required for a cube. The following code is a partial specification of an individual component within this system. The user initially enters in the coordinates of this component upon the Cube (and should never change these afterwards, unless a new puzzle is begun), and the initial colour of this component. At each state transition the user can enter in a rotation, denoted by the `move` input parameter. Based upon the position of this component upon the Cube (indicated by the coordinates entered initially) and the rotation, this component calculates the new colour which would be shown. This requires that the several components which make up a Cube communicate with each other, although we have not shown this here.

```
facet coord((x,y): input (int, int), init-colour: input colour,
            move: input Rotation)
            ::state-based, CubeDefinitions
```

```
//The user enters the coordinates of this component and its
  initial colour. The input 'move' corresponds to any rotation
  a user performs on the Cube, prompting a state change.

 public colouring: [red, blue, yellow, green, white, orange];
 private coordinate: (int, int);
 // The coordinates of this component

 begin
   T0: colouring@s0 = init-colour;
   T1: coordinate@s0 = (x,y);
   T2: coordinate' = coordinate;
   T3: colouring'=getnextcolour(move, colouring);
   //The getnextcolour function defines the new colour which
    is obtained by performing a given rotation

  ...
end coord
```

Any rotation performed by the user will result in several components changing state together, since a rotation affects all the coordinates in a given row or column.

Advantages of the THS definition for Rubik's Cubes

There are some characteristics of a Rubik's Cube which make it appropriate to specify these systems using the Transition History State definition. Firstly, the literal values of the colours at any point are generally unimportant, since the solution techniques are the same no matter what possible combinations of colours the Cubes can come in. Instead, we are concerned only with the particular rotations used, which are distinguished by the components which change state together. There may be multiple rotations which affect a particular component. In a cubic Rubik's cube, for example, one component can be affected by either rotating the relevant row or column. In more complex shapes, there may also be additional rotations which affect certain components. The Transition History State definition is designed to capture these inter-connections.

When specifying a Rubik's Cube with the intent of using a computer to solve puzzles, efficiency is often a consideration. As we see from the specification above, each component is required to calculate the new colour after processing a rotation as input. While in practice this is not a particularly expensive calculation, there are several reasons we might want to identify how many times a given component must perform this calculation. Firstly, in more complex Cube solids (such as spheres, or solids with unusual axes of rotations), some components may be affected by fewer possible rotations than others. Identifying how

many times a given component's colour is altered during each behaviour provides us with information about which axes of rotation are used most frequently. Secondly, this type of puzzle has applications for the area of distributed computing. If a separate processor is used for the calculations of each component, then the components which are affected most frequently require more processor time to perform their calculations. In order to distribute the load evenly, we must track how often a particular component has to perform this calculation, on average. The Transition History State definition implicitly provides such a facility by ensuring that the state of a component is dependent upon its colour *and* the number of state transitions it has undergone.

Rubik's Cubes can also be specified using a Basic State system, as introduced in Chapter 4. This is expected, as we have already noted that a Transition History system can be interpreted as a Basic State system that has been augmented with an additional 'counter' variable tracking state transitions. However, using a Transition History system allows us to emphasise the nature of state transitions of a Cube — namely, that these are determined by a number of components changing state together. Finally, another aspect of Rubik's Cubes which makes them worthy of further study either as a Basic State system or — more appropriately — Transition History system is the complexity of a typical Rubik's Cube puzzle. We will now present one of these puzzles, then show in Section 5.3 how it can be considered as an example of a view update problem, but in terms of system specification. We note that there are already a number of algorithms, techniques, and solvers which are used for Rubik's Cube puzzles. The claim here is not that CCF supercedes any existing work, but rather that this framework can be used to consider a number of contrasting factors when analysing inconsistencies within such a system.

Rubik's Cube Puzzles

The typical Rubik's Cube puzzle asks for the shortest sequence of rotations which produces a particular design upon the faces of the Cube. This description encompasses the typical timed competition to solve a Cube. That is, the shortest sequence of rotations which results in a solved Cube can also be reasonably expected to be the fastest solution. More generally, a Rubik's Cube puzzle is usually similar to the following example.

Q1: ''You are given a cube with an initial colouring as depicted in Figure 5.4. You perform 12 rotations, and are only permitted to observe their effects on the coordinates $1, 3, 6, 25, 33$ and 43. You observe that these coordinates undergo a number of colour changes together and separately:

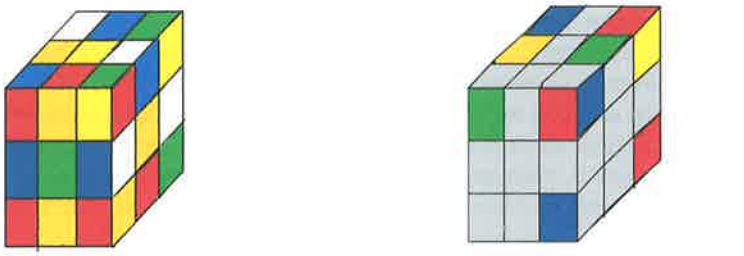

Figure 5.4: An example of a complex Rubik's Cube puzzle

Change Number	Coordinates changed
First change	1, 3
Second change	6, 25, 43
Third change	43
...	...
Seventh change	33, 3, 25, 43

These coordinates end up displaying the colours in shown in Figure 5.4. What were the rotations?
Bonus questions: Can you achieve these changes and final colouring in fewer than 12 movements? What is the shortest solution you can find?''

In this question, the user is provided with information about the starting state of the Cube, and limited information (via the table) about the rotations which are made. From this, the user is asked to deduce what the rotations were, and whether there is a shorter sequence of behaviours which affects the components mentioned in the same way. To answer this question, the user must extrapolate from the behaviour t of a subsystem (the components shown in the table) to deduce the shortest, or *minimal*, behaviour of the underlying system which restricts to t on the subsystem.

Depending on the puzzle, there may be several solutions. Because of this, it is important when setting these questions to be sure that if a solution is claimed to be unique then it is possible to prove that this is the case. However, if there do exist multiple solutions, they often share common sub-sequences. (In fact, memorization of the sub-sequences which are used to achieve certain simple patterns on the Cube faces is considered vital when trying to solve a Cube quickly). Given these common sub-sequences, the natural question is to ask whether there is a unique 'shortest' solution to a puzzle which is contained as a subsequence within every other solution to this puzzle. If so, this shortest solution consists of the necessary and sufficient rotations to solve the puzzle.

Identifying the puzzles for which there exists such a necessary and sufficient solution to a smaller puzzle involving a subsystem is of paramount importance in timed competitions. For example, when solving a Cube, competitors often break the solution up into various subgoals, consisting of swapping the colours of two corners or solving an entire row. If there exist unique minimum solutions to these subgoals, then together these solutions may represent the fastest solution to the overall problem. In the following section, we use CCF to relate the behaviour of a subsystem to the behaviour of the underlying system. From this discussion, we will be able to discuss the mathematics of minimal behaviours such as those which occur in the solutions to Rubik's Cube puzzles.

5.3 The Behaviour of Subsystems

One important question arising from the analysis of a modular system specification is whether it is possible to deduce anything about the system from observing the behaviour of a subsystem. This question can be rephrased as a system specification analog to the database view update problem. The view update problem, discussed in Chapter 2, centres around the question of whether a given view of the database permits propagatable updates. A propagatable update is a view update for which we can deduce the underlying database update.

The Rubik's Cube puzzle of Section 5.2.7 can be interpreted as a view update problem. That is, this puzzle asks the user to deduce the underlying system behaviour, given information about how certain components (a *view* of the system) are affected. Another example of a specification analog to a database update problem occurs when defining an interface between a user and a system. Such an interface identifies a visible subsystem which enables a user to view or deduce certain information about the wider system. For example, a designer may hide implementation details under this interface so that an observer can deduce — from observed behaviour via the interface — *what* is happening in the system, but not *how* it is happening. We return to this example in Chapter 6, where we use an interface to describe information that can be restored after a system failure.

The similarity between database updates and questions of subsystem interactions suggests that solution techniques for the one problem might be used in the other. In order to demonstrate this, we will use CCF to present a formal means of relating subsystems and the underlying system.

5.3.1 CCF And Subsystem Representation

In this section we will show how CCF can be used to formally relate a state of a subsystem P to a state of the wider system. By extension, we can then express the relationship between a behaviour of P and a behaviour of the underlying system. We will use a view functor K to identify those elements of the

system which comprise the subsystem P. More specifically, K will identify a subcategory $\mathbf{C}_V^P$ of the abstract state category $\mathbf{C}_V$ of the system. Objects in $\mathbf{C}_V$ will correspond to those datatypes visible to P, while morphisms in $\mathbf{C}_V$ will correspond to the functions, variables and constants of P.

In Section 4.4.2 we showed how an CCF sketch $(G_V, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$ can be used to represent a state of a system (noting that this holds for both Basic State systems and THS systems, although the sketches are constructed differently). Given a subsystem P of this system, we can also represent P by an CCF sketch, and by extension generate the system category of P. In the following definition, we assume the abstract state category of the underlying system is presented by the CCF sketch $(G_V, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$, as introduced in Definition 5.3 for a THS system.

Definition 5.18. *The* abstract state category *of a subsystem P is the category presented by the CCF sketch $(\mathrm{G}_V^P, \mathcal{D}_V^P, \mathcal{L}_V^P, \mathcal{C}_V^P)$ where*

- *The graph G_V^P consists of those nodes and edges of G_V (of Definition 5.3) which represent system elements visible within P.*
- *The family of pairs of paths $\mathcal{D}_V^P$ are those pairs of paths in $\mathcal{D}_V$ (of Definition 5.3) obtained from the constraints within that part of the data universe visible to P.*
- *The families $\mathcal{L}_V^P$ and $\mathcal{C}_V^P$ are those cones and cocones of $\mathcal{L}_V$ and $\mathcal{C}_V$ (of Definition 5.3) which are comprised solely of elements within G_V^P*

Typically, the entire data universe will be visible to P, and hence we have $\mathcal{D}_V^P = \mathcal{D}_V$, $\mathcal{L}_V^P = \mathcal{L}_V$ and $\mathcal{C}_V^P = \mathcal{C}_V$. If this is the case, then $\mathbf{C}_V^P$ is a subcategory (although not necessarily a full subcategory) of $\mathbf{C}_V$, the abstract state category of the underlying system. We can formalise this by defining the inclusion functor K as follows.

Definition 5.19. *The inclusion functor $K : \boldsymbol{C}_V^P \to \boldsymbol{C}_V$ identifies those data elements of the system which are within the subcategory P.*

We can also consider P itself as a self-contained system. That is, we can construct the category of states of P, when it is considered in isolation from the surrounding system. This means that any system axioms which are not within P and yet constrain elements of P (for example, constraining public variables of P relative to the data universe, or to other variables outside P) do not apply. The techniques of Chapter 4 can be applied to generate the system category for the system P, as well as the interaction consistency category. Using these techniques, we can then obtain the category of states of the system P. If the system in question is a THS system, we denote the category of states of P by $\mathbf{TH\ States}_P$. The analysis we present in this section, however, applies to all types of systems.

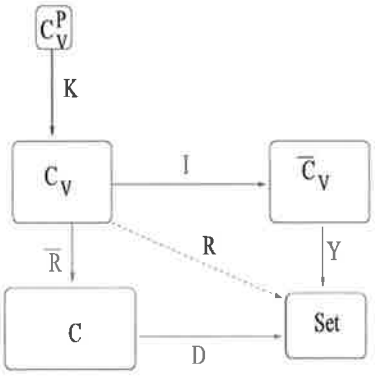

Figure 5.5: The CCF framework corresponding to a system and subsystem P

We can then use the view functor $K : \mathbf{C}_V^P \rightarrow \mathbf{C}_V$ to relate the states and behaviour of the underlying system to the states and behaviour of P. This is done by composition with K, the functor of Definition 5.19. That is

$$K^* : \textbf{TH States} \rightarrow \textbf{TH States}_P$$

Composition of an object $R : \mathbf{C}_V \rightarrow \textbf{Set}$ of **TH States** with K results in an object $R_P : \mathbf{C}_V^P \rightarrow \textbf{Set}$ of $\textbf{TH States}_P$. A morphism $r = (R, [M]_i, R')$ of **TH States** can be composed with K to produce a morphism of $\textbf{TH States}_P$. Informally, composition with K represents a restriction of a system state or behaviour to a state or behaviour of P. Figure 5.5 illustrates this situation using the categories of CCF.

5.3.2 Deducing Underlying Behaviours

We can now use CCF to analyse the relationship between the behaviours of P and the behaviours of the underlying system. This is similar to the task of relating a database view update to an update of the underlying database. To demonstrate this, let T and T' be states of the subsystem P — that is, objects in $\textbf{TH States}_P$. Let $t : T \rightarrow T'$ be a morphism in $\textbf{TH States}_P$, representing a consistent behaviour of the component P. When we examine this behaviour in the context of the wider system, there are two issues which arise.

Firstly, there may be no underlying system behaviour which restricts to this behaviour t. This means that t cannot be observed when P is placed within the environment of the wider system. To avoid this problem, we must restrict our attention to those behaviours of P which are in the image of K^*. Secondly, there may be *several* underlying behaviours of the system which restrict to this behaviour t. In this case, simply observing t is not sufficient for us to deduce the behaviour of the underlying system. However, if all these underlying behaviours

share some common characteristics, then observing t allows us to deduce that these characteristics must have been demonstrated by the underlying behaviour which occurred. One useful example of this is the situation where all system behaviours restricting to t share some common minimal behaviour r. In this case, observing t is sufficient to let us deduce that *at least* the behaviour r must have occurred in the system.

In general, we are most interested in the case where the observation of the behaviour r within the wider system is both necessary and sufficient for us to observe t on the subsystem P. Ascertaining the existence of such a minimal behaviour r is useful in systems such as those modelling a Rubik's Cube, where a user seeks the fastest solution that permits certain observations. In such cases, the fastest solution also corresponds to that solution which is composed of fewest, or least disruptive, statechanges. Another example of where the existence of minimal behaviours is useful occurs when analysing specifications for mechanical systems. In these systems, which include for example engines and thermal systems, each statechange represents an energy transfer of some form. Since each state change corresponds to an expenditure of energy or fuel, we are interested in identifying the shortest sequence of state transitions which provide us with the desired behaviour t.

5.3.3 Minimal Behaviour And Pre-cocartesian Liftings

If there is a certain system behaviour r which is both necessary and sufficient for us to observe t on the subsystem P, then r can be said to be analogous to a pre-cocartesian lifting of t. We define such a minimal underlying behaviour r as follows.

Definition 5.20. *Given a behaviour $t : T \to T'$ of a subsystem P and a functor $K^* :$ **TH States** $\to$ **TH States**$_P$ obtained from the view functor K of Definition 5.19, the* canonical underlying morphism, *or* canonical underlying behaviour *$r : R \to R'$ of the underlying system possesses the following properties*

- $K^*(r) = t$
- *for all $r' : R \to R''$ such that $K^*(r') = t$, there exists a unique r'' such that $r'' \circ r = r'$ and $K^*(r'') = id$.*

These conditions are depicted in Figure 5.6. It should be noted that a canonical underlying behaviour is not guaranteed to exist for every behaviour t.

The importance of such minimal updates has been demonstrated in databases, and briefly discussed in terms of system specification in Section 5.3.2. By using Definition 5.20 to represent minimal underlying behaviours in CCF, we are able to formally characterise those systems in which these behaviours are present.

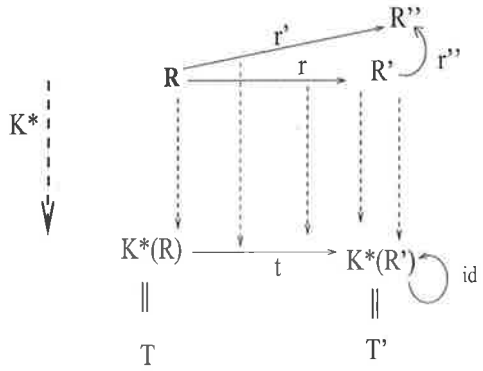

Figure 5.6: An example of a canonical underlying behaviour r for the behaviour t

This increased formality — as opposed to *ad hoc* methods of deducing what may be happening in a partially hidden system — means that designers can guarantee a certain amount of predictability within their specifications when using CCF. We return to this topic in Chapter 6, in which we discuss how a system which is "predictable" to a user can be said to display a certain degree of consistency.

However, Definition 5.20 does not take any issues of consistency into consideration. That is, one or more of the morphisms r, r' and r'' mentioned in the definition might be inconsistent. In general, we are most interested in the existence of such a morphism r in *consistent* systems only. In the following section, we identify some more constraints upon a canonical underlying morphism r, if it is to exist within a consistent system. We also demonstrate how the existence of a pre-cocartesian lifting property allows us to make deductions about the consistency of other behaviours within a system.

5.3.4 Validity of Minimal Behaviours

As we have already established, each morphism r of **TH States** represents a behaviour but not necessarily a *consistent* behaviour. Definition 5.17 describes these morphisms which do represent a consistent behaviour, being those within any subcategory **TH States(D)**. These morphisms are then said to be defined over D, and sometimes referred to as *consistent morphisms*. When analysing a system with the intention of identifying minimal behaviours, we are primarily interested in those pre-cocartesian liftings r which represent consistent behaviours. Restricting the canonical underlying morphisms of Definition 5.20 to those which might occur in consistent systems provides us with the following definition of *application canonical morphisms*.

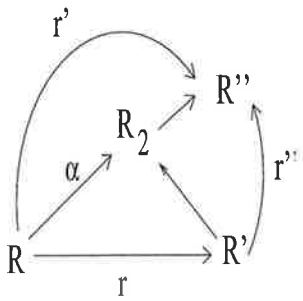

Figure 5.7: Minimal behaviours, from Theorem 5.22

Definition 5.21. *A morphism* $r : R \to R'$ *of* ***TH States*** *is an* application canonical morphism *of* $t : T \to T'$ *if*

- r *is a consistent morphism (as detailed in Definition 5.17), and* $K^*(r) = t$
- *for any consistent morphism* $r' : R \to R''$ *such that* $K^*(r') = t$*, there is a consistent morphism* $r'' : R' \to R''$*, where*
  1. $K^*(r'') = id$ *and* $r'' \circ r = r'$
  2. *there exists at least one system model* D *such that* r'' *and* r *are both within* ***TH States(D)****.*

This definition enforces the minimal behaviour of Definition 5.20 within a consistent system. Definition 5.21 requires that for each consistent r' where $K^*(r') = t$, there exist a consistent r'' such that the sequence of state transitions described by $r'' \circ r$ is identical to the sequence of state transitions described by r' (1). Furthermore, if we observe r within the system, then observe r'', this composite behaviour should be consistent itself (2). While this second condition follows trivially in a Basic State system from the fact that $r' = r'' \circ r$, THS systems are more complex. This is because, in **TH States**, morphisms are distinguished by the components which change state together, not by the values visible in the observed states. Thus, a single THS morphism r' may be defined over several different system models D, and it is possible that none of them permit the observation of the state R'. If this is the case, then condition (2) of Definition 5.21 cannot be satisfied, and hence r will not be an application canonical morphism. We reiterate, however, that this distinction between an application canonical morphism and a canonical underlying behaviour is relevant only when considering THS systems. In Basic State systems the difference between consistent and inconsistent morphisms is not complex enough to affect minimal behaviour conditions. The following discussions and proofs are therefore essential only when discussing THS systems.

Consistency and Pre-cocartesian Liftings

Using application canonical morphisms (Definition 5.21) instead of the simpler pre-cocartesian property of canonical underlying morphisms (Definition 5.20) involves relaxing some constraints upon a pre-cocartesian lifting while strengthening others. Informally, in order for r to be an application canonical morphism it must first represent a consistent behaviour. Thus, not every canonical underlying morphism of t will necessarily be an application canonical morphism of t. This is because Definition 5.20 does not require that canonical underlying morphisms represent consistent behaviours.

On the other hand, when considering an application canonical morphism r, we only require that certain morphisms r' factor through r. The morphisms r' for which this is required by Definition 5.21 are the r' which restrict to t and are consistent. By contrast, Definition 5.20 requires that *every* morphism r' restricting to t (consistent or not) must factor through the canonical underlying morphism r. Thus, not every application canonical morphism of t will necessarily be a canonical underlying morphism.

Finally, for an application canonical morphism r, we require that r and r'' (for the morphism r'' satisfying $r' = r'' \circ r$, for a consistent morphism r restricting to t) are both valid over some common system model D. For a canonical underlying behaviour, this requirement of validity does not apply. Thus, this is a stricter condition on application canonical morphisms than on pre-cocartesian liftings.

5.3.5 Deductions from Application Canonical Morphisms

In this section, we discuss some implications of using a canonical underlying morphism in analysis (Definition 5.20) instead of an application canonical morphism (Definition 5.21), and vice versa. As a result of this process, we will eventually show that a canonical underlying morphism, if consistent, is always an application canonical morphism. This means that we can use the much simpler canonical underlying morphism property without concern over whether the system in question is consistent.

We begin by proving that if r is a canonical underlying morphism of t, and if r' is a consistent behaviour such that $K^*(r') = t$, then r' is defined only over those system models D which r is also defined over.

Theorem 5.22. *Let $r = (R, [\hat{M}]_i, R')$ be the canonical underlying morphism of $t : T \to T'$ and let $r' = (R, [M]_j, R'')$ be a morphism in some* ***TH States(D)*** *such that $K^*(r') = t$. Then r is defined over D.*

Proof. From Definition 5.20, r' must factor through r since r is a canonical underlying morphism. Since composition in **TH States** is concatenation of the relevant tuples, it follows that $[M]_j$ contains the list $[\hat{M}]_i$. That is,

$$[M]_j = [\hat{M}]_i \bigcup [N]_j$$

for some additional list $[N]_j$ of transitions, and therefore

$$r' = (R, [[\hat{M}]_i, [N]_j], R'')$$

We can now define the behaviour $\alpha = (R, [\hat{M}]_i, R_2)$, where R_2 is the state which results from observing $[\hat{M}]_i$ over the system model D. α is then defined over D, and this situation is depicted in Figure 5.7. We now claim that $R_2 = R'$, and prove this by first showing that $K^*(R_2) = T'$.
Since $r' = (R, [\hat{M}]_i, [N]_j, R'')$, and $K^*(R'') = T'$, if $K^*(R_2)$ were *not* to equal T', then the transitions of $[N]_j$ must imply some statechanges of components within the subsystem P. However, because r is the underlying canonical morphism of t, there exists a morphism r'' such that $r' = r \circ r''$. Given the families of transitions comprising r and r', and the definition of composition in a THS system, we can conclude that

$$r'' = (R', [N]_j, R'')$$

However, the definition of a canonical underlying morphism also requires that $K^*(r'') = id$. Since the number of transitions undertaken at any point distinguishes state, then the existence of even a single transition within r'' which affects the subsystem P would mean that $K^*(r'') \neq id$. Therefore $[N]_j$ cannot contain any transitions affecting P. Thus, we can deduce that

$$K^*(R_2) = T'.$$

As a result of this, the morphism $\alpha = (R, [\hat{M}]_i, R_2)$ (defined over D and shown in Figure 5.7) must restrict to t — that is, $K^*(\alpha) = t$. Since $r = (R, [\hat{M}]_i, R')$ is the underlying canonical morphism of t, α must factor through r. Clearly, the only way this can be achieved, given the definition of composition in **TH States**, is with the identity morphism. That is,

$$\alpha = id \circ r$$

Thus, α is r, the pre-cocartesian lifting of t, and therefore $r = \alpha$ is defined over D. □

Moreover, since $r' = (R, [\hat{M}]_i, [N]_j, R'')$, and is defined over D it follows that $(R_2, [N]_j, R'')$ is also defined over D. Furthermore, since $\alpha = r$, we know that $R_2 = R'$, and therefore that r'' is also defined over D.

This gives rise to the following lemma.

Lemma 5.23. *If there exists a canonical underlying morphism r of some behaviour t, then any consistent behaviour r' restricting to t will only be valid over those system models D for which*

- *r is defined and*
- *some r'' is defined, where $r' = r'' \circ r$ and $K^*(r'') = id$*

This Lemma states that r and r'' are defined over a common system model D, which is precisely the criteria required for a consistent r to be an application canonical morphism.

Theorem 5.24. *If r is a consistent canonical underlying morphism, then it is also an application canonical morphism.*

Proof. This follows from Theorem 5.22 and Lemma 5.23 □

In fact, since the proof of Theorem 5.22 did not rely on consistency of r, then even if r is not itself consistent, it will still act as an application canonical morphism. As a result, we can simply use canonical underlying morphisms (of Definition 5.20) in systems where analysis of minimal behaviours will be undertaken. Because Definition 5.20 is phrased in terms of pre-cocartesian liftings (with no properties specific to system specification), this means that any existing work involving pre-cocartesian or cartesian liftings may be adapted within CCF for analysis of minimal behaviours. That is, we have shown that the system specification analog to the database view update problem is not further complicated by the addition of such considerations as consistency issues.

5.3.6 Cofibrations and Application Canonical Morphisms

When analysing a system with the intention of identifying minimal behaviours, it is a natural extension to consider the existence of a canonical update, or minimal underlying behaviour, for multiple behaviours $t : T \to T'$ of a subsystem P. This leads to an examination of situations in which the functor K^* is a cofibration.

Definition 5.25. *The functor K^* is a cofibration when for any behaviour $t : T \to T'$ in the subsystem P, for any system state R such that $K^*(R) = T$, there exists a cocartesian lifting $r : R \to R'$ of t.*

In general, if r is a pre-cocartesian lifting of t then it does not follow that r is necessarily a cocartesian lifting. Nevertheless, the utility of cocartesian liftings is often greater than that of pre-cocartesian liftings, simply because they allow us to deduce much more about the system. This can be seen in a specification context also. For example, if there exists an application canonical morphism r for every behaviour $t : T \to T'$ of P, we can deduce that, to a large extent, P controls or predicts the behaviour of the subsystem. Composition of underlying canonical morphisms then allows us to obtain the least disruptive system behaviour, given any consistent series of observations in the subsystem. That is, we can deduce the least amount of work that must be performed by the system in order for certain observations on the subsystem to be made. This is especially useful in those systems where a state-change represents a physical transfer of energy, such as a mechanical system.

Furthermore, we can apply existing category theoretical constructions involving cocartesian liftings and fibrations to the categories of states of systems, without risking the introduction of inconsistent behaviours. This is because

any cocartesian lifting r is also a pre-cocartesian lifting, and we have shown in Section 5.3.5 that any pre-cocartesian lifting of a morphism t will also act an application canonical morphism of t. In other words, any conclusions about cocartesian liftings will apply equally well to application canonical morphisms, and do not rely on the existence of possibly inconsistent behaviours within a system.

5.3.7 Minimum Behaviours and System Applications

The identification of situations in which a minimal behaviour exists has many applications in terms of system specification. One of these is proving the existence of a unique fastest solution to a Rubik's Cube puzzle (Q1, Section 5.2.7), and identifying this solution. In Q1, the user was provided with information about the behaviour of a subsystem of the Rubik's Cube. Specifically, the question details how the subsystem consisting of components $1, 3, 6, 25, 33, 43$ behaves, in terms of the ordering of each state transition of these components. This behaviour we term t. We note that the Rubik's Cube is specified using a THS system specification, so the only information provided is the relative ordering of component state transitions. That is, we are told when these components change colour, but not what colour they change to.

Q1 then asks for the shortest system behaviour which provides this behaviour t on the subsystem. More accurately, it first asks for the system behaviour which allows the observation of t and consists of 12 rotations. The bonus question then asks if there is a shorter system behaviour allowing the observation of t. If there is a pre-cocartesian lifting r of the behaviour t, where r consists of more than 12 movements, then this problem will have no solution. On the other hand, the pre-cocartesian lifting of t is the *unique* shortest behaviour restricting to t. That is, if a solution can be shown to be the pre-cocartesian lifting of t, it is guaranteed that there will be no shorter solution. In this way, the claim of a unique solution can be proved.

These liftings are already used, albeit informally, in Rubik's Cube tournaments. Competitors in these tournaments traditionally memorize the movements which achieve certain subgoals. For example, the shortest sequence of movements which swaps the colours of two coordinates is usually memorized, as is the sequence of movements which results in a row of coordinates displaying the same colour. These memorized behaviours are the pre-cocartesian liftings of the required observations upon the subsystem which consists of two corners, or a row of the Cube. By combining these memorized movements, a player can generate the shortest solution to a more complex observed behaviour t. In this case, the player is making use of a fibration.

When discussing minimal behaviours in this section, we have restricted our attention to the Transition History State definition. This was primarily because

the motivating example, a Rubik's Cube puzzle, is specified using this definition. However, the same analysis of subsystem behaviours can be performed for any system, as there is no reliance upon any of the assumptions of the Transition History State definition. In the following section, we present another definition of state, the Input State definition. System specifications of both this and the Basic State type can also benefit from the exploration of minimal behaviours.

5.4 The Input State Definition

In both Basic State systems and Transition History systems the components communicate with each other via shared data. In Rosetta, this communication mechanism is implemented via the **public** keyword, which declares variables visible to other facets. Multiple facets can then apply constraints to data shared in this way, thereby interacting with each other. In this section, we examine systems which use a different method of communication. Specifically, we will examine how communication can be achieved using message passing, by means of input/output parameters.

A component may receive input from a user, or from another component within the system. For example, the alarm clock of Appendix A receives input from the user when the time is being set. However, in the same example the two facets `clockBeh` and `alarmBeh` receive input from each other.

Definition 5.26. *Two components `f1` and `f2` are* associated *components if they communicate by means of input/output parameters.*

The behaviours of associated components must synchronise if the system is to be consistent. In this chapter, we present a formal framework within which we can ensure that this synchronisation takes place. The analysis of systems which communicate in this way merits a detailed discussion because input parameters are generally treated differently from ordinary variables. The framework we present makes it possible to formally recognise these differences, some of which we discuss briefly below.

Firstly, in the majority of systems a component is not compelled to act upon input. This means that a change in the value of an input parameter should not cause the component in question to undergo a state transition. This demonstrates a difference between the message passing and data sharing communication mechanisms, because changing the value of a shared variable may compel other components to change state. This was illustrated by the definition of a morphism in **TH States**, wherein we use the data sharing communication mechanism. Secondly, a component may not constrain its own input. That is, while a component may specify a range of acceptable values for an input parameter, it may not further constrain this parameter relative to state. Finally, when describing behaviours in systems which use input parameters, it is standard practice to label state transitions with the value of input, whereas the

state transitions we have encountered thus far have been unlabelled. The following section describes how we can generate categories within CCF to describe a system which uses input parameters.

5.4.1 Characteristics of an Input State system

In practice, systems generally use both message passing and shared variables to communicate. We will therefore consider input/output parameters as an extension to the existing characteristics of a system. That is, both Transition History and Basic State systems may be extended to use input parameters, and the existing characteristics of these systems augmented with the necessary properties to support these parameters. For the sake of convenience, we will confine our attention in this section to the alterations which must be made to the characteristics of a Basic State system in order to allow input parameters. The following informal definition describes the properties of a Basic State system which has been augmented to make use of input parameters. In subsequent sections we will formalise each point of this definition.

Definition 5.27. *An **Input State** system is one possessing the following characteristics:*

1. *A component's state is distinguished by the current values of its variables, except input parameters as detailed in (5) below*
2. *The values in the next state are constrained relative to the values in the current state only and input values.*
3. *The value of variables in the initial state may be constrained relative to constants or functions applied to constants.*
4. *Transitions from one state to another are labelled with the input value (if any) which prompted this transition. Behaviours are distinguished both by these labels and by the states which are observed.*
5. *The value of input being offered does not determine the current state of the component.*
6. *A system cannot constrain its own input beyond the constraints which apply in every state.*

A state of such a system will be referred to as an Input State, and a behaviour of this system as an Input State behaviour. The first four characteristics are those of the original Basic State system. The fifth characteristic implies that simply offering input is not sufficient to compel a component to process this input. However, when input is accepted, the fourth characteristic ensures that the subsequent state transition is labelled with this input. If a state transition does not correspond to any input (for example, the only component which has changed state is one which does not possess any input parameters), we will label this transition with δ. Stutters will be labelled with ϵ, although the analysis we perform will be stuttering-insensitive.

5.4.2 An Example Input State System

We will use the following Rosetta code throughout this section as an example of an Input State system.

```
facet f1(x: input int):: state-based
  private y, q: int;
  begin
   T0: q@s0=0;
   T1: y = x+1;
   T2: q' = q+1;
end f1

facet f2(x: output int) :: state-based
  int x;
  begin
   L1: x@s0 = 0;
   L1: x' = x-2;
end f2
```

In this code the variable x of component `f2` provides the input for component `f1`. This is usually reflected in some language-dependent syntax — in Rosetta, we indicate this by using identical input/output parameter names.

5.4.3 The CCF Sketch Representing An Input State System

To represent an Input State system in CCF, we first generate the system category $\mathbf{C}$, representing the entire dynamic system.

Definition 5.28. *The* system category *for an Input State system is the category $\boldsymbol{C}$ presented by the CCF Sketch $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ where*

- *G is the directed graph of Clarification 4.4, used in Definition 4.9 to generate the system category of the underlying Basic State system*
- *$\mathcal{D}$ is the family of pairs of paths in G discussed in Clarification 4.5, and also used in Definition 4.9*
- *$\mathcal{L}$ is the family of cones in G constructed according to Construct 4.6, and also used in Definition 4.9*
- *$\mathcal{C}$ is the family of cocones in G constructed according to Construct 4.7, and also used in Definition 4.9*

This CCF sketch is constructed identically to the CCF sketches of Transition History and Basic State systems, although the constraints (represented by $\mathcal{D}$) of an Input State system may be different. Specifically, in an Input State system a component will not constrain its own input. Input parameters, however, are

represented within this sketch identically to other variables. A model of this category $\mathbf{C}$ is a functor $D : \mathbf{C} \rightarrow \mathbf{Set}$ which preserves finite limits and countable coproducts.

5.4.4 The CCF Sketch Representing An Input State

We will use an abstract state category $\mathbf{C}_V$ to assist with identifying a single Input state. This category will be constructed identically to the abstract state category of a Basic State system, and we later show how to augment this with input variables such that Definition 5.27 is satisfied.

Definition 5.29. *The* abstract state category *for an Input State system is the category $\mathbf{C}_V$ presented by the CCF sketch $(G_V, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$ where*

- *G_V is the directed graph discussed in Clarification 4.13 and used in Definition 4.16 to generate the abstract state category of the underlying Basic State system*
- *$\mathcal{D}$ is the family of pairs of paths in G_V discussed in Clarification 4.14, and also used in Definition 4.16*
- *$\mathcal{L}_V$ and $\mathcal{C}_V$ are the families of cones and cocones in G_V, constructed according to Construct 4.15, and also used in Definition 4.16*

This CCF sketch is constructed identically to the CCF sketch defining the abstract state category for a Basic State system. Again, this implies that input parameters are implemented identically within this sketch to other variables. The abstract state category for the code of Section 5.4.2 is the lower category shown in Figure 5.8.

A model of $\mathbf{C}_V$ is a functor $R : \mathbf{C}_V \rightarrow \mathbf{Set}$ preserving finite limits and countable coproducts. However, such a functor for an Input State system does *not* represent a state of this system. We discuss this in more detail below.

5.4.5 Modelling Input Parameters In CCF

Definitions 5.28 and 5.29 do not reflect any of the additional characteristics with which the Basic State system has been augmented to allow for input parameters. Of the characteristics given in Definition 5.27, the only relevant one at this stage is the fifth — namely, the value of an input parameter should not determine the current state of the component. Specifically, this means that the value of an input parameter should be hidden, so that the only time it has a visible effect upon the component is if input is processed. We can achieve this by defining a subsystem consisting of all those data elements of the system which are *not* input parameters. An Input state will then be a state of this subsystem.

Definition 5.30. *The subsystem I consists of all those elements of an Input State system which are* not *input parameters.*

The concept of defining subsystems in this way was introduced in Section 5.3, wherein we discussed interfaces and the action of hiding variables.

Following the notation introduced in Section 5.3, we will represent this subsystem I by a subcategory $\mathbf{C}_V^I$ of the abstract state category. $\mathbf{C}_V^I$, known as the *input abstraction category*, will be generated from the CCF sketch $(\mathrm{G}_V^I, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$.

Definition 5.31. *The* input abstraction category $\boldsymbol{C}_V^I$ *is the category presented by the CCF sketch* $(\mathrm{G}_V^I, \mathcal{D}_V^I, \mathcal{L}_V, \mathcal{C}_V)$*, where*

- G_V^I *is the graph obtained by removing the edges representing input parameters of any component from the graph* G_V *of Definition 5.29.*
- *The families* $\mathcal{D}_V, \mathcal{L}_V$ *and* $\mathcal{C}_V$ *are those families of the same names in Definition 5.29.*

This construction of the families $\mathcal{D}_V, \mathcal{L}_V$ and $\mathcal{C}_V$ is consistent with the composition of the graph G_V^I, since input parameters (which are not represented in G_V^I) can neither be declared within the data universe, nor constrained relative to the data universe (and $\mathcal{D}_V, \mathcal{L}_V$ and $\mathcal{C}_V$ refer only to the data universe). That is, the subsystem I observes the same data universe as the wider system. Owing to the composition of the CCF sketch in definition 5.31, it is clear that the input abstraction category $\mathbf{C}_V^I$ will always be a subcategory of the abstract state category (although not necessarily a full subcategory). We formalise this by the following inclusion functor.

Definition 5.32. *The* input inclusion functor $In : \boldsymbol{C}_V^I \to \boldsymbol{C}_V$ *is a view functor (as in Definition 5.19) which identifies those data elements of the system which are* not *input parameters of any component.*

That is, In identifies the elements of the subsystem I of Definition 5.30. In therefore acts as an interface, hiding those variables whose values should not be used to distinguish state. The input inclusion functor for the code of Section 5.4.2 is shown in Figure 5.8.

We can use the input abstraction category to define a state of an Input State system:

Definition 5.33. *A state of an Input State system is a functor* $R_{In} : \boldsymbol{C}_V^I \to$ ***Set*** *preserving finite limits and countable coproducts.*

A state R_{In} of an Input State system is therefore not dependent upon the value of input parameters, as these are not represented within the input abstraction category $\mathbf{C}_V^I$.

5.4.6 Consistency of Input System states

Not every state $R_{\mathrm{In}} : \mathbf{C}_V^I \to \mathbf{Set}$ of an Input State system will be a consistent state of this system. To establish consistency of an Input state, we use

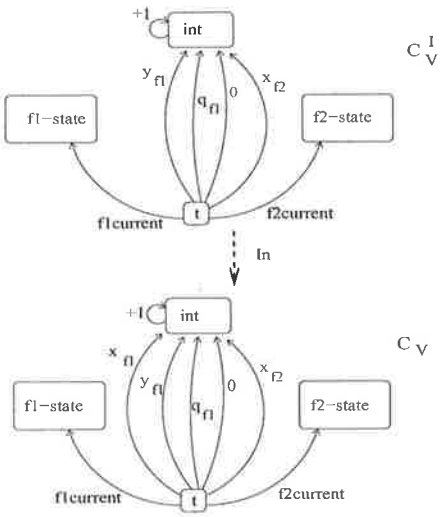

Figure 5.8: $\mathbf{C}_V^I$ does not include input parameters, unlike $\mathbf{C}_V$.

the critera for component-wise and interaction-wise consistency established in Definitions 4.18 and 4.19. Component-wise consistency of an Input state is then given by the following definition.

Definition 5.34. *An Input state R_{In} is component-wise consistent if*

$$R_{In} = R \circ In$$

where In is the input inclusion functor of Definition 5.32, and R is a component-wise consistent Basic state, according to the conditions of Theorem 4.21.

Applying Definition 4.19 (interaction-wise consistency) to Input states requires more care, however. This is because, unless a component is processing input, the values of its input parameters are irrelevant to consistency. For example, the input parameter x of `f1` in Section 5.4.2 need not have the same value as the output parameter x of `f2`, if `f1` chooses not to process input. However, if `f1` *does* choose to process input then interaction-wise consistency will require that these have the same value, to ensure that the input processed by `f1` is the same as that provided by `f2`. To satisfy these two alternative consistency conditions, we first define an interaction consistency category for an Input State system. This category $\bar{\mathbf{C}}_V$ implements the consistency requirements of the underlying Basic State system, and is therefore identical to the interaction consistency category of the underlying Basic State system.

Definition 5.35. *The* interaction consistency category *for an Input State system is the category $\bar{\mathbf{C}}_V$ presented by the CCF sketch $(G_V, \bar{\mathcal{D}}_V, \mathcal{L}_V, \mathcal{C}_V)$ where*

- *G_V is the directed graph of Definition 4.22, used in Definition 4.24 to generate the interaction consistency category of the underlying Basic State system*
- *$\mathcal{D}_V$ is the family of pairs of paths in G_V of Definition 4.22, representing the axioms of the data universe and also used in Definition 4.24*
- *$\bar{\mathcal{D}}_V$ is the family of pairs of paths in G_V, detailed in Clarification 4.23, and also used in Definition 4.24*
- *$\mathcal{L}_V$ and $\mathcal{C}_V$ are the families of cones and cocones of G_V, detailed in Definition 4.22 and also used in Definition 4.24*

$\mathcal{D}_V$ represents the constraints of the data universe, while $\bar{\mathcal{D}}_V$ represents the requirement that shared variables must have the same value when accessed from any component. This applies to input/output parameters as well, since under the Basic State definition these are treated no differently to ordinary variables. For example, for the code of Section 5.4.2, the constraint $x_{f1} = x_{f2}$ would be represented in $\bar{\mathcal{D}}_V$. Thus, as mentioned earlier, the interaction consistency category of Definition 5.35 for an Input State system is identical to the interaction consistency category of Definition 4.24 for the underlying Basic State system, were input parameters treated as ordinary variables.

However, input parameters are *not* ordinary variables according to Definition 5.27, and hence this equality between associated output and input parameters need not apply in every Input state. To formally represent the 'part-time' nature of this constraint, we define the the following two degrees of interaction-wise consistency of an Input state. The first will be required when a component chooses to process input, while the second (weaker) will be required when a component is not processing input and hence the value of its input parameters are irrelevant to interaction-wise consistency. In Section 5.4.7 we will define precisely when each degree of interaction-wise consistency is required.

Definition 5.36. *An Input state R_{In} is* interaction-wise consistent *if*

$$R_{In} = R \circ In$$

where In is the input inclusion functor of Definition 5.32 and R is an interaction-wise consistent Basic state, as given by the conditions of Theorem 4.25.

This definition requires that associated input/output variables are equal, even if they are not represented in $\mathbf{C}_V^I$, by phrasing interaction-wise consistency in terms of the underlying Basic State system.

Definition 5.37. *An Input state R_{In} is* input interaction-wise consistent *if there exists a functor $Y : \bar{\boldsymbol{C}}_V \rightarrow \boldsymbol{Set}$ such that*

$$R_{In} = Y \circ In^I$$

where $In^I : \boldsymbol{C}_V^I \rightarrow \bar{\boldsymbol{C}}_V$ is a quotient functor as shown in Figure 5.9.

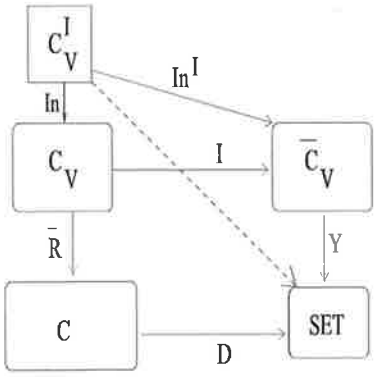

Figure 5.9: Consistency of an Input state is an extension of the Basic state consistency of Figure 4.4

This second definition does not enforce the equality of input/output variables, since these are not mentioned in $\mathbf{C}^I_V$. Figure 5.9 shows the commutativity required for each type of consistency. We may usefully compare this to Figure 4.4, which depicts consistency of the underlying Basic State system. As before, a *valid* Input state R_{In} is one which preserves finite limits and countable coproducts.

5.4.7 The Category of Input System States

In order to examine the relationship between states of an Input State system, we will construct the category **Input States**. Objects in this category will be consistent states of an Input State system, while morphisms will represent behaviours of the system. In order to satisfy the description of an Input State system in Definition 5.27, morphisms will consist of tuples describing the states observed throughout a behaviour, as well as the labels α upon each state transition, where these labels describe the input values of parameters.

Definition 5.38. *The category* ***Input States*** *is comprised of*

- *objects which are those functors* $R_{In} : \boldsymbol{C}^I_V \rightarrow \boldsymbol{Set}$ *which preserve finite limits and countable coproducts and satisfy Definitions 5.34 and 5.37.*
- *morphisms which are tuples* $r = (R_{In}, [\alpha, R_{In}]_i, (\alpha', R'_{In}))$ *of objects in* ***Input States****, satisfying Property 5.39 (detailed below as the Input State Ordering Property) and where this tuple demonstrates consistency considerations as given below in Clarification 5.40.*

Two objects R_{In} and R'_{In} in **Input States** will represent equal states if and only if they are naturally isomorphic. In the following section, we show how this definition provides us with morphisms which represent Input State behaviours,

and discuss why the differing degrees of consistency throughout a morphism are required.

Morphisms in the category of Input States

A morphism $r = (R_{\text{In}}, [(\alpha, R_{\text{In}})]_i, (\alpha', R'_{\text{In}}))$ represents the Input states R_{In_i} observed, and the labels α_i upon each state transition. Each state transition label α consists of a list of components which change state during a particular transition, as well as the input parameter and value which is being processed by each of these component. If a component changes state without processing input (such as component `f2` in Section 5.4.2), the input parameter and value for this component are given the null value δ.

As we have seen when discussing the categories **States** and **TH States**, not every sequence of states represents a consistent behaviour. The sequences which do are those in which the states obey some ordering. Property 4.28 was used to enforce this in the category **States**; here we present an adaptation of this property to Input State systems.

Property 5.39 (The Input State Ordering Property). *A tuple* $r = (R_{In}, [(\alpha, R_{In})]_i, (\alpha', R'_{In}))$ *satisfies the* Input State Ordering Property *if the following conditions hold:*

- $[R_{In}]_i$ *is a list of Input states formed by composition of* In *with a list* $[R]_i$ *of Basic states obeying Property 4.28 (the State Ordering Property for Basic State systems)*
- *For each state transition label* α_i*, if the system is in the state* $R_{In_{i-1}}$ *and each component processes the input listed in* α_i*, the system will be in the state* R_{In_i}.

This property can be used to identify those tuples r which define a sequence of states which are ordered correctly (that is, according to the `finext` function which orders the `fi-state` datatype for any component `fi`).

However, there is another requirement on the states which are observed during r. Specifically, if a state transition is prompted by a component choosing to process input, the values of the associated input/output parameters must synchronise. This means the states of the components which receive and provide this input must demonstrate interaction-wise consistency rather than just input interaction-wise consistency. This stricter degree of consistency ensures that the input received is equal to the input provided by the associated component.

Clarification 5.40. *Morphisms in the category* ***Input States*** *of Definition 5.38 are tuples* $r = (R_{In}, [\alpha, R_{In}]_i, R'_{In})$ *satisfying Property 5.39 and where for each label* α_i *in* r *where* α_i *mentions a component* fj*, the subsystem consisting of* fj *and* associated *components demonstrates the interaction-wise consistency of Definition 5.36 in the state* $R_{In_{i-1}}$

That is, whenever a component processes input we identify the subsystem consisting of this component and its associated components, as described in Definition 5.26. We then use the techniques of Section 5.3 to obtain the category of states of this subsystem. We will then require that whenever a component is processing input, the subsystem consisting of this component and its associated components must be in a state which demonstrates interaction-wise consistency. That is, various subsystems must display differing degrees of consistency throughout a morphism r.

The identity morphism in **Input States** is a morphism $i = (R_{\text{In}}, [\delta], R_{\text{In}})$, indicating that no input has been processed by any component. Equality of morphisms is equality of the tuples up to conflation of consecutive identical elements $[\delta, R_{\text{In}}]$ in each list $[\alpha, R_{\text{In}}]_i$. We remember that a labelling of δ means that no input has been processed.

Valid Input State System Behaviours

We have already seen in the **States** and **TH States** categories that some morphisms r are not defined over a single system model D. This means that the behaviour represented by r is not consistent, since the axioms relating one state to another are not then satisfied. To identify those morphisms representing consistent behaviours, we defined subcategories **States(D)** and **TH States(D)**. We follow the same reasoning here.

Definition 5.41. ***Input States(D)*** *is the subcategory of* ***Input States*** *associated with the system model D. It is comprised of*

- *objects which are those Input states $R_{In} : \boldsymbol{C}_V^I \to \boldsymbol{Set}$ for which*

$$R_{In} = D \circ \bar{R} \circ In$$

  for some valid Basic state identification functor $\bar{R} : \boldsymbol{C}_V \to \boldsymbol{C}$ (as in Definition 4.20), and where In is the input inclusion functor of Definition 5.32

- *morphisms which are those morphisms $r = (R_{In}, [\alpha, R_{In}]_i, R'_{In})$ of* ***Input States*** *where every R_{In_i} in this tuple is an object within* ***Input States(D)***.

5.4.8 Analysing Input State systems

The category **Input States** provides a formal framework for analysing a system which makes use of input parameters. This category can be used in the same way as the categories **States** and **TH States** introduced earlier. Specifically, we can identify inconsistencies in individual states or behaviours by examining the objects and morphisms of **Input States**. Once a particular inconsistency has been detected, we can use the taxonomy of inconsistencies created in Section 4.7 to compare methods of resolution.

However, to ensure that analysis using CCF (particularly for the more complex Input State systems) has validity in a wider context, it is necessary to compare CCF to existing analysis frameworks. One of these frameworks, which we will examine in the rest of this chapter, is provided by the **Span(Graph)** category. Using this category enables us to identify co-existing behaviours of interacting components, by making use of such constructs as pullbacks in **Graph**. In the following section, we will show how the dynamic aspects of CCF and structural aspects of the category of theories (Chapter 3) can be used together to provide an analog to the analysis which can be performed using **Span(Graph)**. This means that we can translate specifications between the CCF and **Span(Graph)** frameworks when performing system analysis.

5.5 Span(Graph) and CCF

Span(Graph) is the bicategory of spans of graphs, introduced in Section 2.4.1. It can be used to analyse processes or components within a system. We represent each process within a system by a graph depicting its states and state transitions. Pullbacks in **Graph** are used to compute horizontal composition of morphisms in **Span(Graph)** and can be used to identify behaviours of individual processes which should co-exist. Further categorical constructions upon the bicategory **Span(Graph)** allow us to perform other analysis tasks, such as finding minimal behaviours.

Span(Graph) is a particularly useful tool for analysing those systems which communicate via message passing, or input/output parameters. This is because these systems characterise state transitions at a more detailed level than the majority of systems we have examined so far. That is, each state transition of a component is labelled, where the labels represent input provided by an associated component. **Span(Graph)** recognises these different labels upon a transition as different labels upon the edges of the graph representing a process or component.

As we have mentioned before, we can make use of pullbacks in **Graph** (or alternatively, morphisms in **Span(Graph)**) to identify synchronised behaviours of associated components. For example, two components `f1` and `f2` might communicate via some variable x. Most commonly, this will be an input parameter for the component `f2`, where this input is provided by `f1`. These components will be represented by graphs G_{f1} and G_{f2} respectively. We can then represent the entire system by the graph G which is the pullback in **Graph** of G_{f1} and G_{f2}, respecting this shared variable x. Owing to the composition of **Span(Graph)**, G can also be obtained by horizontal compositions of morphisms within this category. This situation is shown in Figure 5.10. Paths in each graph represent behaviours of the component in question. Thus, we see that a path in G represents those behaviours of `f1` and `f2` which can coexist, given that they

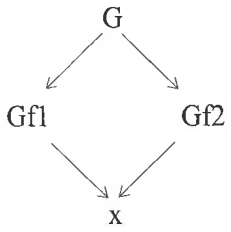

Figure 5.10: G, the pullback in **Graph**, represents the system in which `f1` and `f2` communicate via x

communicate via x. That is, G consists of paths which make Figure 5.10 commute (and in fact, specifically is the pullback in **Graph**). Section 2.4.1 discusses this in more detail.

In the rest of this chapter, we discuss how we can use the dynamic aspects of CCF and the structural aspects of the category of theories to also identify coexisting behaviours. Given two components `f1` and `f2` which interact as described above, we will generate a *representational component* describing this interaction. This will be done by using the techniques of Section 3.4.2, in which representational components were first introduced. We will then translate this representational component from the category of theories into the categories of CCF. This enables us to identify the behaviours of the representational component, whereas within the category of theories we could only identify the structure. Behaviours of the representational component will then correspond to coexisting behaviours of the original components (in this case, `f1` and `f2`). That is, a behaviour of the representational component describes a behaviour of the consistent system.

We emphasise this by generating a graph from the category of states of each component. The graphs generated from the category of states of `f1` and `f2` will be G_{f1} and G_{f2} respectively. We then prove that the graph generated from the category of states of the representational component will be G. Finally, we show that the graph morphisms of Figure 5.10 can be obtained from the theory morphisms used to generate the theory of the representational component. In this way, we will perform the same analysis in CCF as could be performed by using **Graph** or **Span(Graph)**. The resulting graphs and graph morphisms can then be used for further analysis in the more familiar **Span(Graph)** framework.

5.5.1 CCF and Representational Components

In Section 3.4.2 we first introduced the concept of a *representational component*. This was defined to be the component which represents a particular interaction between other components. Definition 3.16 describes how we can construct

the representational component, given knowledge about the structures of the interacting components `f1, ..., fn` and their communication. That is, by using this definition we can structurally relate the representational component to the original components within the category of theories. In this section we will show how, given communicating components `f1, ..., fn`, we can use CCF to relate the behaviours of these to each other in terms of the behaviours of the representational component. Framing the relationships in terms of behaviours in this way is essential for relating the analysis performed using CCF to that performed using **Span(Graph)**.

Specifically, given the techniques of Section 3.4.2, we will use the category of theories to generate the representational component of the `f1, ..., fn` interaction. We will then model the representational component using CCF. Finally, we will relate behaviours of all these components to each other, and identify those behaviours which can co-exist in a consistent system.

In order to generate the representational component, we require information about how `f1, ..., fn` communicate. That is, we need to know what information is shared, or which variables are used for message passing. Variables which are shared between multiple components of `f1, ..., fn` will be mapped to the *same* element of `Rep`. For example, if `f1` and `f2` communicate via a shared variable x or via a pair of input/output variables $f1.x$ and $f2.x$, then there will be a single variable in `Rep` representing x. In Section 3.4.2, this information was provided by the designer and modelled using a virtual theory $Th_{xy_1,...,xy_n}$. The use of this virtual theory allowed us to identify the theory Th_{Rep}, of the representational component `Rep` by using colimit diagrams in the category of theories. In the rest of this section we will show how to represent `Rep` in CCF, and examine its behaviours in relationship to the behaviours of components `f1, ..., fn`.

We begin by defining a category $\mathbf{C}$ which will represent the system consisting of the interacting components `f1, ..., fn` and the representational component `Rep`. That is, $\mathbf{C}$ is the *system category* for the system `f1, ... fn, Rep`. $\mathbf{C}$ is also known as the *complete representational category* of this interaction.

Definition 5.42. *The* complete representative category $\mathcal{C}$ *of the interaction between components* `f1, ..., fn` *(where* `Rep` *is the representational component of this interaction) is the category presented by the CCF sketch* $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ *where*

- G *is a directed graph wherein the nodes represent the datatypes declared in any of the components* `f1, ..., fn`*, and the edges represent the variables, functions and constants declared in any component* `f1, ... fn` *and* `Rep`*.*
- $\mathcal{D}$ *is a family of pairs of paths in* G *with common source and target, where each pair is obtained from a constraint within any of the components* `f1, ..., fn` *and* `Rep`*.*

- *$\mathcal{L}$ and $\mathcal{C}$ are families of cones and cocones in G, representing product datatypes and constructively specified datatypes respectively.*

We note that $\mathcal{D}$ does not include any constraints relating variables of `Rep` to the variables of the original components `f1, ..., fn`. Additionally, we note that the variables, functions and constants of a component `fi` are represented distinctly from the associated variables, functions and constants of `Rep`. However, we represent the datatypes of `Rep` as being the same as those datatypes of the components `f1, ..., fn`. That is, each datatype is represented once only.

It is obvious that in order to construct this sketch, we require knowledge of the structure and constraints which make up `Rep`. This information is obtained by constructing Th_{Rep} in the category of theories, as described in Section 3.4.2. The complete representative category for the code specifying switches in Section 3.5.1 can be seen in Figure 5.11. We then construct the *abstract state category* $\mathbf{C}_V$, which represents the data visible in a single state of the system `f1, ..., fn, Rep`. This abstract state category is obtained by using Definition 4.16 in conjunction with this system.

We have already stated that $\mathbf{C}$ does not represent any of the constraints which relate `Rep` to the original components `f1, ... fn`. Because these constraints enforce shared variables, they are represented in CCF by defining an *interaction consistency* category $\bar{\mathbf{C}}_V$. $\bar{\mathbf{C}}_V$ is obtained by using Definition 4.24 in conjunction with the system `f1, ..., fn, Rep`. The following remark emphasises some important properties of the resultant interaction consistency category.

Remark 5.43. *Equalities in the interaction consistency category $\bar{\mathbf{C}}_V$ enforce communication between components `f1, ..., fn, Rep`. This includes the overlap, or interference between variables of the original components `f1, ..., fn` and variables of `Rep`.*

For the **Switch Example** code of Section 3.5.1, this means the equality

$$switch_{f1} = switch_{f2} = switch_{Rep}$$

is represented in $\bar{\mathbf{C}}_V$.

Together, the complete representative category $\mathbf{C}$, the abstract state category $\mathbf{C}_V$ and the interaction consistency category $\bar{\mathbf{C}}_V$ can be used to construct the diagram in Figure 4.4 for the system `f1, ..., fn, Rep`. Given these categories, we can apply the techniques of Chapter 4 to construct **States**, the category of states of the system `f1, ..., fn, Rep`. Morphisms in **States** represent behaviours of this system `f1, ..., fn, Rep`, while morphisms in **States(D)** (for any system model D) represent *consistent* behaviours. That is, a morphism r within some subcategory **States(D)** corresponds to behaviours of the components `f1, ..., fn, Rep` which can coexist if this system is to be consistent. For more detail on these subcategories we refer the reader to Chapter 4.

We can then use a view functor $I_{Rep}{}^*$ to define the subsystem of `f1, ..., fn, Rep` consisting solely of the component `Rep`. The technique of using view functors to identify subsystems was introduced in Section 5.3, and requires us to identify a subcategory $\mathbf{C}_V^{Rep}$ of the abstract state category. $\mathbf{C}_V^{Rep}$ should represent the variables, constants, functions and datatypes visible to the representational component `Rep`. Because the data universe for the representational component is the same as for the components `f1, ..., fn`, $\mathbf{C}_V^{Rep}$ is clearly a subcategory of $\mathbf{C}_V$ (although not necessarily a full subcategory). We formalise this by the inclusion functor below.

Definition 5.44. *$\boldsymbol{C}_V^{Rep}$ is the* abstract state category *of `Rep`, as described in Definition 5.18. The functor $I_{Rep} : \boldsymbol{C}_V^{Rep} \rightarrow \boldsymbol{C}_V$ identifies those data elements of the system `f1, ..., fn, Rep` which comprise the representational component `Rep`.*

This functor is shown in Figure 5.12. As described in Section 5.3, composition with I_{Rep} allows us to describe the restriction of a system state or behaviour to the subsystem `Rep`. That is, if $\mathbf{States}_{Rep}$ is the category of states of the representational component `Rep`, then

$$\mathrm{I}^*_{Rep} : \mathbf{States} \rightarrow \mathbf{States}_{Rep}$$

describes this restriction. In practice, because of the way in which `Rep` is constructed, every morphism in $\mathbf{States}_{Rep}(\mathbf{D})$ will be obtained by applying I^*_{Rep} to a morphism within **States(D)**. That is, any consistent behaviour of the component `Rep` must correspond to consistent and synchronised behaviours of the components `f1, ..., fn`. This is because any constraint within a component `fi` is also present within `Rep` and acts upon the data elements of `Rep` associated with the relevant data elements of `fi`. Section 3.4.2 explains this in more detail.

Although we indicate by the names of the categories (eg. **States**) that this is a Basic State system, the conclusions of this chapter apply to all types of systems.

5.5.2 Statespaces of Components

In the previous section, we showed how to relate communicating components `f1, ..., fn` to the representational component `Rep` of this interaction within the framework provided by CCF. We then used the view functor $I_{Rep}{}^*$ of Definition 5.44 to relate $\mathbf{States}_{Rep}$ (the category of states and behaviours of `Rep`) to **States** (the category of states of the system `f1, ..., fn, Rep`). In this section, we will show how to explicitly relate $\mathbf{States}_{Rep}$ to the category $\mathbf{States}_{fi}$ of states of `fi`, for any `fi` in `f1, ..., fn`. This will enable us to formally identify those behaviours of `f1, ..., fn` which synchronise, or can coexist. Finally, we translate these results into the category of graphs, to show how these results are analogous to those obtained using **Span(Graph)**.

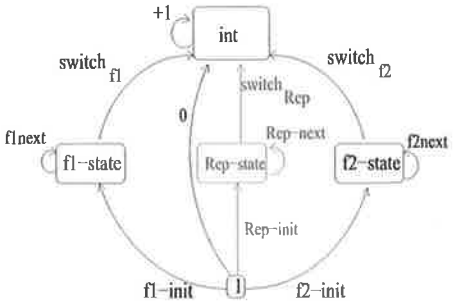

Figure 5.11: The complete representative category **C** for the code in Section 3.5.1

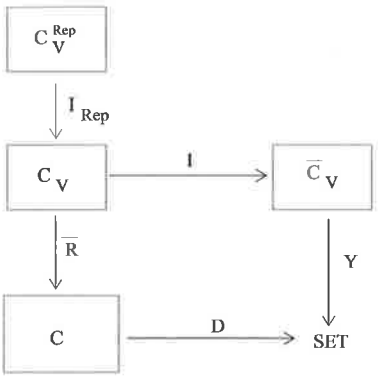

Figure 5.12: The functor I_{Rep} identifies elements in `Rep`

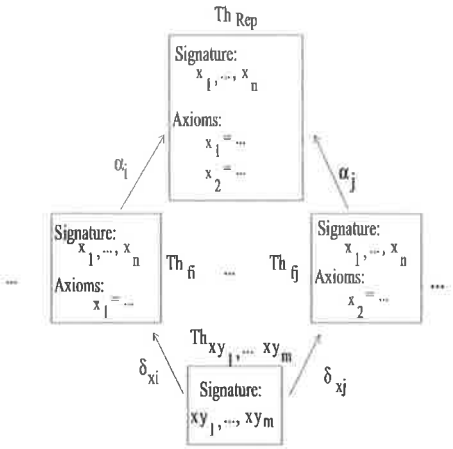

Figure 5.13: The theory $Th_{Rep} = Th_1 \times ... \times Th_{fi} \times_{xy_1,...,xy_m} Th_{fj} \times ... \times Th_n$ is the colimit of theories $Th_1, ...Th_n$ and $Th_{xy_1,...xy_m}$

First we return to the techniques introduced in Section 3.4.2, which relate the theories of components `f1, ..., fn`. In Section 3.4.2 we introduced the concept of a 'virtual' theory $Th_{xy_1,...,xy_m}$. This identifies the variables x_k and y_k, from two components `fi` and `fj` respectively of `f1, ..., fn`, which are intended to interfere in this interaction. If Th_{Rep} denotes the theory of the representational component `Rep`, and $Th_{f1}, ..., Th_{fn}$ denote the theories of components `f1, ... fn`, then Th_{Rep} is the colimit in the category of theories of the Th_{fi}s and the virtual theories $Th_{xy_1,...,xy_k}$, for any two components `fi` and `fj` which demonstrate interference. This situation is shown in Figure 5.13, where the theory morphisms $\alpha_1, ...\alpha_n, \delta_{x1}, ...\delta_{xk}$ are used to express the relationship between these theories. Figure 3.3 shows a concrete example of this type of interaction.

To translate these theory morphisms into relationships between behaviours of components, we will first define $\mathbf{States}_{fi}$, for each `fi` in `f1, ... fn`. This denotes the category of states of component `fi`, when `fi` is considered in isolation from the rest of the system. Following this, we define $\mathbf{States}_x$, the category of states of a 'virtual' theory representing shared variables. We then show how to translate the theory morphisms $\alpha_1, ...\alpha_n, \delta_{x1}, ..., \delta_{xk}$ into functors in CCF. We use these functors to relate the categories $\mathbf{States}_{fi}$, $\mathbf{States}_x$ and $\mathbf{States}_{Rep}$. Finally, we translate these categories and functors into **Graph** and show that they are precisely the same as those objects and morphisms used in **Graph** to identify coexisting behaviours.

The category States$_{fi}$

We will use the techniques of Chapter 4 to construct the category of states of each component `fi` in `f1, ..., fn`. That is, we examine the specification of `fi` and from this, create the system category $\mathbf{C}_{fi}$, the abstract state category $\mathbf{C}_V^{fi}$, and the interaction consistency category $\bar{\mathbf{C}}_V^{fi}$ of `fi`. The abstract state category $\mathbf{C}_V^{fi}$ of `fi` represents the data visible in a single state of the component `fi`. According to the discussions in Section 5.3, this will be a subcategory of the abstract state category $\mathbf{C}_V$ of the system `f1, ... fn, Rep` (although not necessarily a full subcategory).

Remark 5.45. *$\mathbf{C}_V^{fi}$, the abstract state category of `fi`, can be described as a subcategory of the abstract state category $\mathbf{C}_V$ of the system `f1, ... fn, Rep`, using Definition 5.18.*

These categories $\mathbf{C}_{fi}$, $\mathbf{C}_V^{fi}$, and $\bar{\mathbf{C}}_V^{fi}$ together form a diagram as depicted in Figure 4.4, denoting a consistent state of `fi`. From this, Section 4.6 describes how we would obtain the category of states of this system consisting solely of `fi`.

Definition 5.46. *The category **States**$_{fi}$ is the category of states of the component `fi`, when this is considered as a complete system in itself.*

The category States$_x$

In Section 3.5, we introduced the concept of a 'virtual' theory $Th_{xy_1,...,xy_m}$ representing the information shared between two components `fi` and `fj` in `f1, ..., fn`. This enabled us to use colimits to denote the representational component Th_{Rep} of this interaction. We described in that section how $Th_{xy_1,...,xy_m}$ consists of the data universe and declarations of a single variable xy_k for each pair of variables x_k and y_k from `fi` and `fj` respectively, where these variables are intended to interfere in this interaction. The virtual theory imposes no constraints upon these variables xy_k.

We will now formulate a CCF sketch representing the theory $Th_{xy_1,..,xy_m}$. That is, this will be the CCF sketch representing a new 'virtual' component which simply declares variables $xy_1, ..., xy_m$ but does not constrain them in any way:

```
facet virtual:: state-based
  xy1, ..., xym: int;
end virtual
```

This component `virtual` does, however, include the data universe of the system `f1, ..., fn`.

From this CCF sketch, we use the techniques of Chapter 4 to generate the system category $\mathbf{C}_x$, abstract state category $\mathbf{C}_V^x$, and the interaction consistency category $\bar{\mathbf{C}}_V^x$ of the system consisting solely of this component `virtual`. Once again, we note that the abstract state category $\mathbf{C}_V^x$ of `virtual` represents the data visible in a single state of the component `virtual`. This will be a subcategory of the abstract state category $\mathbf{C}_V$ of the system `f1, ... fn, Rep`, just as the category $\mathbf{C}_V^{fi}$ is.

Remark 5.47. *$\mathbf{C}_V^x$, the abstract state category of the virtual component represented by $Th_{xy_1,\ldots,xy_m}$, can be described as a subcategory of the abstract state category $\mathbf{C}_V$ of the system `f1, ... fn, Rep`, using Definition 5.18.*

We can then follow the techniques of Chapter 4 to generate the category $\mathbf{States}_x$ of states of the system consisting solely of `virtual`. We note that this will be an infinite, connected category, since `virtual` lacks any constraints upon the variables declared within it.

We now describe how the categories $\mathbf{States}_x$, $\mathbf{States}_{fi}$ and $\mathbf{States}_{Rep}$ (discussed in Section 5.5.1) can be related to each other. This requires translating the theory morphisms which relate the theories $Th_{fi}, Th_{xy_1,\ldots,xy_m}$ — for all the 'virtual' theories representing interference between different pairs — and Th_{Rep} into functors between the abstract state categories $\mathbf{C}_V^{fi}$, $\mathbf{C}_V^x$ and $\mathbf{C}_V^{Rep}$.

Relating the statespaces of components

Figure 5.13 depicts the theory morphisms

$$\alpha_i : Th_{fi} \rightarrow Th_{Rep} \text{ and } \delta_{x_i} : Th_{xy_1,\ldots,xy_m} \rightarrow Th_{fi}$$

which we use to define Th_{Rep} by a colimit diagram in the category of theories. In this section we translate these theory morphisms into functors between the abstract state categories $\mathbf{C}_V^{Rep}$, $\mathbf{C}_V^x$, and $\mathbf{C}_V^{fi}$ of Definition 5.44 and Remarks 5.47 and 5.45.

Firstly, for each component `fi` we create a functor V_{fi} relating the abstract state category $\mathbf{C}_V^{fi}$ of component `fi` to the abstract state category $\mathbf{C}_V^{Rep}$ of the representative component `Rep`. We create each functor V_i by translating the theory morphism

$$\alpha_i : Th_{fi} \rightarrow Th_{Rep}$$

which models the information given by the designer about shared variables to a functor

$$V_{fi} : \mathbf{C}_V^{fi} \rightarrow \mathbf{C}_V^{Rep}$$

The action of V_{fi} is defined in the following construct.

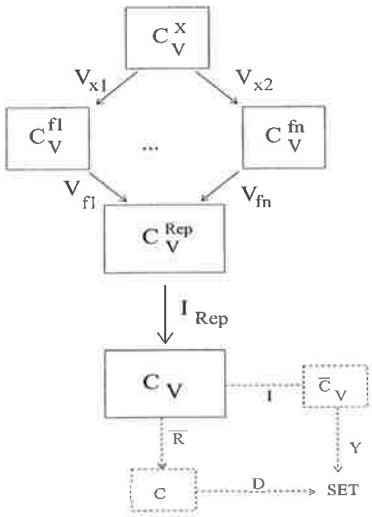

Figure 5.14: The solid lines indicate the CCF representation of Figure 5.13

Construct 5.48. *For each component* `fi` *of* `f1, ... fn`*, the functor* V_{fi} : $C_V^{fi} \rightarrow C_V^{Rep}$ *is defined on objects and morphisms as follows.*

- V_{fi} *takes an object in* C_V^{fi} *representing a datatype* `type` *(visible to* `fi`*) to the object in* C_V^{Rep} *representing the datatype in* `Rep` *determined by the action of the theory morphism* $\alpha_i : Th_{fi} \rightarrow Th_{Rep}$ *on* `type`*.*

- V_{fi} *takes a morphism in* C_V^{fi} *representing a constant, function or variable* `x` *(visible to* `fi`*) to the morphism in* C_V^{Rep} *representing the constant, function or variable in* `Rep` *determined by the action of the theory morphism* $\alpha_i : Th_{fi} \rightarrow Th_{Rep}$ *on* `f`*.*

A similar construct defines the action of the functor $V_{xi} : \mathbf{C}_V^x \rightarrow \mathbf{C}_V^{fi}$, where $Th_{xy_1,\ldots,xy_m}$ defines the interaction between `fi` and some `fj`. That is, the action of V_{xi} is determined by the theory morphism $\delta_{xi} : Th_{xy_1,\ldots,xy_m} \rightarrow Th_{fi}$ which identifies those variables of `fi` which interfere with the variables of some other component `fj` during this interaction.

Figure 5.14 depicts the interaction of these categories and functors. This is the CCF equivalent of the relationships shown in Figure 5.13.

We can use these functors V_{fi} and V_{xi} to relate behaviours of `Rep` to behaviours of the components `f1, ..., fn`. This is achieved by using the techniques of Section 5.3 to obtain functors

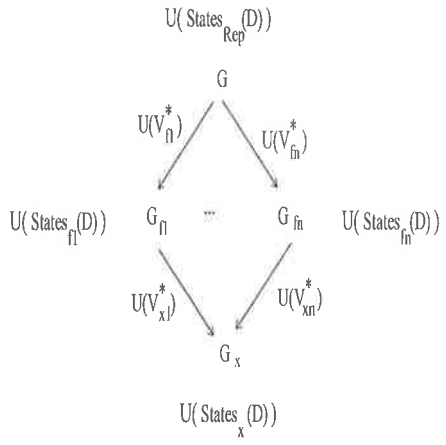

Figure 5.15: The graphs resulting from the CCF analysis

$$\mathrm{V}^*_{fi} : \mathbf{States}_{Rep} \rightarrow \mathbf{States}_{fi} \text{ and}$$
$$\mathrm{V}^*_{xi} : \mathbf{States}_{fi} \rightarrow \mathbf{States}_{x}$$

Informally, V^*_{fi} describes the restriction of a behaviour r of `Rep` to the behaviour which `fi` displays during r. Similarly, V^*_{xi} describes the restriction of a behaviour r of `fi` to the behaviour displayed by the virtual theory during r. Because `Rep` is more highly constrained than each `fi` — which are in turn more highly constrained than the virtual theory — these functors are well-defined. That is, any consistent behaviour of `Rep` describes a consistent behaviour of `fi`, since all constraints upon `fi` also apply to `Rep`. In turn, consistent behaviours of `fi` describe consistent behaviours of the virtual theory.

These functors ${V_{fi}}^*$ and V^*_{xi} therefore enable us to relate the behaviours of the representational component `Rep` directly to the behaviours of the interacting components `f1, ..., fn`. To show the similarities between these categories and functors and the entities used in **Span(Graph)** to identify co-existing behaviours, we make use of the forgetful functor

$$U : \mathbf{Cat} \rightarrow \mathbf{Graph}$$

We will apply this functor to the categories $\mathbf{States}_{Rep}(\mathbf{D})$, $\mathbf{States}_{fi}(\mathbf{D})$ and $\mathbf{States}_{x}(\mathbf{D})$ (for every $\mathbf{States}_{x}(\mathbf{D})$ representing interference between a pair of components `fi` and `fj`), as well as to the functors V^*_{fi}: $\mathbf{States}_{Rep} \rightarrow \mathbf{States}_{fi}$ and V^*_{xi} $\mathbf{States}_{fi} \rightarrow \mathbf{States}_{x}$. In this way we obtain graphs representing the statespaces of the components `fi, ..., fn`, the component `Rep` and all the virtual theories, as well as graph morphisms relating these statespace graphs. We show these graphs and morphisms in Figure 5.15.

The following theorem describes the precise relationship between these graphs and, by extension, between the consistent behaviours of `Rep` and components `f1, ..., fn`. This relationship is precisely that established by using **Graph** or **Span(Graph)** when identifying co-existing behaviours of `f1, ..., fn`.

Theorem 5.49. *The graph G obtained by applying U to every category $\mathbf{States}_{Rep}(D)$ (ie. for every system model D) is the pullback in* **Graph** *of the graphs $G_{f1}, ..., G_{fn}$ (obtained by applying U to the subcategories $\mathbf{States}_{f1}(D)$, ..., $\mathbf{States}_{fn}(D)$) and all the graphs G_x (obtained by applying U to all the categories $\mathbf{States}_x(D)$, each of which represents the states of a virtual theory defining interference between some pair of components).*

Proof. A path $(e_1, ...e_n)$ in G represents a consistent behaviour of `Rep` and therefore corresponds to a morphism r in some $\mathbf{States}_{Rep}(D)$. As discussed in Section 5.5.1, any such morphism can be expressed as a composition of the functor I_{Rep} of Definition 5.44 with a morphism m of **States(D)**. That is, m represents a consistent behaviour of the system `f1, ..., fn, Rep`. This means that m describes morphisms $m_{f1}, ..., m_{fn}$ in the categories $\mathtt{States}_{f1}\mathtt{(D)}$, ..., $\mathbf{States}_{fn}\mathbf{(D)}$, where these behaviours can coexist in a consistent system. Given that the functors $V_{fi} : \mathbf{C}_V^{fi} \rightarrow \mathbf{C}_V^{Rep}$ describe the restriction of a behaviour of `Rep` to the behaviours of `f1, ..., fn` which it implies, we can conclude that

$$\mathrm{V}^*_{fi}(r) = m_{fi}$$

Furthermore, if m_{fi} represents a consistent behaviour of `fi`, then it must describe a consistent behaviour of the 'virtual' component defining the interference between `fi` and some component `fj`. That is, $\mathrm{V}^*_{xi}(m_{fi})$ is a morphism in $\mathbf{States}_x\mathbf{(D)}$. The fact that the behaviours $m_{f1}, ..., m_{fn}$ can coexist means that they all describe the *same* behaviour of the virtual component. That is, $\mathrm{V}^*_{xi}(m_i) = \mathrm{V}^*_{xi}(m_j)$.
By applying U to the composition of these functors V^*_{fi} and V^*_{xi}, we see that the path $(e_1, ...e_n)$ in G is such that Figure 5.15 commutes. This is the case for any such path in G. That is, G is the pullback of graphs $G_{f1}, ..., G_{fn}$, with respect to G_x. □

In summary, the specification of `Rep` is created by using colimits in the category of theories, and its behaviour is represented (via CCF) by using limits in the category of graphs. The resultant analysis of coexisting behaviours yields the same results as that analysis performed using **Span(Graph)**. That is, the structure of Figures 5.15 and 5.10 are identical. As a result, any analysis obtained from CCF (such as that involving the taxonomy of inconsistencies of Section 4.7) can be translated into the more familiar framework of **Span(Graph)**. Equally, existing results from system analysis performed using **Span(Graph)** and other bicategories can be expressed in terms of CCF.

5.5.3 CCF and Problem Solving

In this chapter we have examined different types of systems, in order to show how CCF can be used to solve problems beyond the classification of inconsistencies. We first introduced the Transition History State definition, which is used to represent those systems where the state of the system is dependent upon the number of state transitions undergone. Behaviours in a Transition History system are characterised by those components which must change state together to preserve consistency. An example of this is a Rubik's Cube, where each rotation of the Cube affects a number of components.

We then discussed how this state definition could be used to solve or address problems of the same general type as Rubik's Cube puzzles. Specifically, a Rubik's Cube puzzle provides information about the behaviour of a subsystem (such as the colours of this subsystem). It then requires the user to determine the shortest, or fastest, system behaviour which corresponds to these observations. Other problems of this type involve finding the most efficient way in which a mechanical system can perform a particular task, bearing in mind that in such systems a state transition often corresponds to energy expenditure. We identified the similarities between these types of problems, and the database view update problem. Finally, by making use of the Transition History State definition and constructions within CCF, we showed how solutions to these problems are characterised. This included an exploration of the consistency implications when a pre-cocartesian lifting exists for certain observations t upon a subsystem. Since these liftings are used to analyse the database view update problem, this approach means that we can relate any CCF analysis to database analysis.

We then discussed a third type of system, the Input State system. These are systems in which communication is achieved by the use of message passing, implemented as input/output parameters. These systems can be defined by augmenting a previous system type with input parameters. We created a representation of Input State systems within CCF, thereby formalising the changes which must be made to a system to allow for this new type of communication. We then presented an alternative analysis framework, **Span(Graph)**, which is particularly appropriate for analysing systems in which communication is achieved by message passing. Finally, we showed how the analysis techniques of CCF are analogous to those of **Span(Graph)**, particularly in the area of identifying synchronised behaviours. This means that any analysis undertaken in CCF has validity in this other framework.

When presenting the Input State definition, we found it useful to introduce "degrees" of consistency, requiring some states to demonstrate *interaction-wise consistency*, and others to demonstrate *input interaction-wise consistency* only. We will return to the concept of degrees of consistency in the next chapter, wherein we discuss the final type of inconsistency examined in this thesis. This

is *system failure inconsistency*, and is caused by external failure, rather than specification error. This external failure results in a loss of information within the system. Even though the state of the system can be restored, the restoration may not be accurate, thus leading to potential inconsistencies. We will show how, in certain cases, being able to restore *some* of the state information correctly is sufficient. We characterise a sufficently acceptable restoration as one which provides some pre-defined degree of consistency.

Chapter 6

System Failure Considered as Inconsistency: A Case Study

6.1 System Failure and Its Impact

In previous chapters, we have confined our attention to inconsistencies which arise as a result of errors in the system specification. Inconsistencies of this form are fundamental and reproducible, and will occur in every implementation of this specification. They can be detected and classified using the categories of CCF, and their analysis gives rise to the taxonomy of inconsistencies in Section 4.7.

However, there is another type of inconsistency which can affect the performance of a system. These inconsistencies are caused by some unexpected external error, rather than an incorrect specification. The most common cause of an error of this type is system failure, such as physical damage or a power loss, which causes the internal state of the system to be lost. Although an attempt to restore state can be made after the damage is fixed, an inadequate restoration attempt may cause the system to display inconsistent behaviour. A typical restoration process involves deducing what information was lost, and restoring the closest approximation to this missing data. A question which then naturally arises is how similar the original and restored states must be for the restoration to be deemed adequate.

In some cases it is sufficient to restore an equivalent state, especially when the system is designed for user interaction. In these cases, restoring an equivalent state means restoring a state which appears identical, to the user, to the state in which system failure occurred. While it is possible for a restoration to restore an inconsistent state, the examination of this would require a more detailed specification than the ones CCF is intended to model. As a result, we will assume that the restored state is always consistent, and concentrate instead on the behavioural inconsistencies that may arise from an inadequate state restoration.

6.1.1 System Failure and Blackbox Specification

When specifying an interactive system, a designer typically divides system information into that which is visible to the user and that which is hidden. This division is implemented by means of an *interface*, which hides certain information about system state to an external observer. Implementation details, for example, are often hidden. These systems are sometimes known as blackbox systems, and their state is determined by both the visible and hidden variables.

An interface can be modelled by a view, or inclusion, functor as we introduced in Definition 5.19. If two states are equivalent under an interface, then — from the perspective of an external observer — the differences between the hidden values in each of the states, and the differences between the future visible behaviours from each state, are irrelevant. The construction and use of an interface to hide implementation details is well-established. However, such an

interface also identifies the data which can be restored correctly after system failure. Specifically, when system failure occurs, a user can restore the data that he has been able to observe. The data hidden under the interface has not been observed, so this cannot necessarily be restored correctly. However, the fact that the user cannot see this data (while also meaning he cannot restore it correctly) means that he will accept an incorrect restoration, provided the state restored is equivalent (to him) to the state in which the failure occurred.

The idea that the data which is currently visible can be restored, while hidden data (which might include data recording previous states) is lost differs from the most common definition of recoverability. The usual approach, when analysing system failure, is to assume that the variables which can be restored correctly are those stored in memory. The values which are lost are the current values, which are not yet processed or saved. However, the systems we will analyse here are those which are externally observed. The observer may consist of other systems, a human observer, or even external sensors which can be used to note the physical orientation of objects. The observations made by such an observer can then be used to accurately restore the data that was visible at the time of the system crash. By contrast, the data hidden under the interface cannot be restored, since it was not observed by any other system.

Failure tolerance is the ability of a system to recover completely and correctly from system failure. In Section 6.4.4 we demonstrate how various weaker degrees of failure tolerance can still be useful. In general, complete failure tolerance is a very stringent condition, which cannot be always satisfied. However, less strict conditions are often sufficient to guarantee a minimal degree of consistency when considering essential or critical behaviours. We first introduced the concept of a critical behaviour or subsystem in Section 4.7, when discussing the severity of inconsistencies. We return to this concept here by allowing users to specify that a particular critical subsystem remain unaffected by system failure. In this case, we can define a less strict degree of failure transparency which guarantees the subsystem recovers "well enough" from this failure.

6.1.2 Overview of CCF and System Failure

In this chapter, we will discuss how CCF can be used to analyse the behaviours resulting from system failure. We also discuss how an incorrect state restoration can cause particular types of behavioural inconsistency, and we present some methods of ameliorating or avoiding these inconsistencies. In Section 6.2 we extend the characterisation of Basic State systems to include an interface which hides some values from an observer. This will be modelled by using an abstraction mechanism, or an inclusion functor as introduced in Definition 5.19, to represent the interface.

In Section 6.2.5 we then define the category of states of a system viewed under such an interface, bearing in mind that we now consider equality of states as equivalence under the interface. We define two classes of morphism within this category, one which models the usual system behaviours and one which models the restoration of visible values after system failure. In this way, the one category can be used to analyse both the visible consistent behaviours of the Basic State system under an interface, and the behaviours which result from incorrect restoration of state after system failure.

Section 6.3 discusses the general implications of failure in a system containing subsystems which have been deemed essential. We require that these subsystem continue to function normally even after an incorrect restoration of state following system failure. Section 6.3.1 presents an example of such a system, where the essential subsystem in question is the alert notification component of a security system. We also introduce some constraints on this example system, and describe two fundamental behaviours which are common in security systems. One of these behaviours involves user input to change security settings, and the other details how the system should respond to an alert.

Section 6.4 then discusses ways in which system failure can be analysed using CCF. This requires defining some new constructions on the category of states developed in Section 6.2. Section 6.4.4 extends these constructions to provide a formal definition of the different degrees of failure tolerance using CCF. These degrees of failure tolerance refer to the permitted level of inconsistency displayed by a critical subsystem in the presence of system failure. We make use of the concepts introduced in Section 5.3, in that they express these degrees by constraining the behaviour of the underlying system in terms of the behaviour of the essential subsystem. The higher the degree of failure tolerance of a subsystem, the more we can deduce about its behaviours and the behaviours of the underlying system in the event of system failure.

Section 6.5 explores how user input to the security system is affected by system failure. Specifically, we examine the implications for the user of requiring a strict degree of failure tolerance within the essential alert notification subsystem. To do this, we first show in Section 6.5.1 that enforcing failure tolerance for this subsystem does not have unwanted effects, such as constraining the possible implementations of the system, or conflicting with any of the previously specified system requirements. That is, we show that failure tolerance can be implemented into systems without the need for rewriting entire specifications. We also discuss a further effect of failure tolerance, being the way it can be used to identify ambiguities in an informal specification.

Section 6.6 discusses how the process of refining a system can affect the degree of failure tolerance displayed by this system. This is illustrated by the same security system example of Section 6.3.1. We show how refining the two

fundamental behaviours of Section 6.3.1 can result in a violation of the strictest failure tolerance conditions. We contrast this with some of the weaker degrees of failure tolerance introduced in Section 6.4.4, showing how these can satisfy consistency requirements while still allowing for refinement of a system. Finally, Section 6.7.3 discusses how the specification of particular behaviours in terms of system failure relates to the frame problem.

6.2 The Failure Tolerance State Definition

In this section we extend the definition of a Basic State system to include an interface. This interface is implemented by using a view functor to identify the variables visible under the interface. While the systems we examine will retain the characteristics of the Basic State definition, there are some changes which must be made to implement equality of states under an interface. In the process, we will use the following definition of equivalence.

Definition 6.1. *Two states $s1$ and $s2$ of a system are* equivalent *under an interface if $s1$ and $s2$ appear visibly identical under this interface, and for any trace originating at $s1$ there is a visibly identical trace originating at $s2$.*

It is also possible to apply an interface to systems displaying other characteristics, such as Input State systems or Transition History State systems. The following discussion therefore acts as a template for exploring the issues associated with behavioural equivalence in any system.

6.2.1 Characteristics of the Failure Tolerance State Definitions

The characteristics of the Failure Tolerance State definition are given below.

Definition 6.2. *A* Failure Tolerance *system specification is a Basic State system which is partially hidden under an interface and satisfies the following properties:*

1. *A state is distinguished by the current values of variables visible under the interface and the values visible in all accessible future states of each component.*
2. *Behaviours are distinguished by the visible values observed in each state.*

A *Failure Tolerance* state is a state of such a system, and a *Failure Tolerance* behaviour is a behaviour of such a system. This is intended to be an informal definition only, and we will formalise each point throughout this chapter.

Each Failure Tolerance system can be obtained by hiding part of a Basic State system under an interface. This means each Basic state R of the underlying Basic State system is associated with a Failure Tolerance state. This Failure

Tolerance state will represent R and all visible traces of each component, where these traces originate at R. According to Definition 6.1, this means two Basic states will generate equal Failure Tolerance states if and only the Basic states are equivalent under the interface. The connection between Basic State systems and Failure Tolerance systems has the following consequence:

Remark 6.3. *Under a trivial interface which hides nothing, a Basic State system is identical to a Failure Tolerance system.*

6.2.2 The CCF sketch representing a Failure Tolerance system

As we have discussed above, we obtain a Failure Tolerance system by hiding part of a Basic State system under an interface. More precisely, we use an inclusion, or view, functor as in Definition 5.19 to identify those system elements visible under the interface. This suggests that we should represent a Failure Tolerance system in CCF by a system category $\mathbf{C}$ identical to that used to represent the underlying Basic State system.

Definition 6.4. *The* Failure Tolerance system category $\boldsymbol{C}$ *is the category presented by the CCF sketch* $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ *of the underlying Basic State system, as given in Definition 4.3.*

The elements of this tuple are detailed in Clarifications 4.4 and 4.5 and Constructs 4.6 and 4.7. The Failure Tolerance system category is identical to the system category of the underlying Basic State system.

6.2.3 The CCF sketch representing a Failure Tolerance state

To obtain a categorical representation of a Failure Tolerance state, we must represent not only the current visible values of variables, but any possible future visible values — that is, the possible visible behaviours originating at this state. While we will eventually include the interface, modelled as a view functor described in Section 5.3, we first demonstrate how to represent current and future values using CCF. This requires generating an abstract state category for the Failure Tolerance system which includes a representation of these future values. In the following definition, we assume the system category $\mathbf{C}$ is presented by the CCF sketch $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ as given in Definition 6.4.

Definition 6.5. *The* abstract state category $\boldsymbol{C_V}$ *for a Failure Tolerance system is the category presented by the CCF sketch* $(G_V, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$ *where*

- G_V *is a directed graph, wherein nodes represent datatypes and edges represent variables, functions and constants*
- $\mathcal{D}_V$ *is the family of pairs of paths in* G_V *constructed as detailed in Clarification 4.14, and used in Definition 4.9 to generate the abstract state category of a Basic State system*

- *$\mathcal{L}_V$ is the family of cones detailed in Construct 4.15 and also used in Definition 4.9*
- *C_V is a family of cocones in G_V representing constructively-specified datatypes within the Failure Tolerance system*

Most of these elements are similar to the elements of the CCF sketch used in Definition 4.9 to generate the abstract state category of a Basic State system. The following clarifications discuss those elements which differ.

Clarification 6.6. *The graph G_V of Definition 6.5 is similar to the graph G used in Definition 6.4, with the exception that the edge `e:1 -> fi-state` which in G represents the `fi-init` function (for each component `fi`), represents in G_V the `fi-current` function. That is, G_V of Definition 6.5 represents the `fi-current` function for any component `fi`, but not the `fi-init` function.*

Structurally, G_V is identical to G. However, the labellings on edges is different, indicating that G_V will be used to identify the current and future values of variables. G, by contrast, is used to identify *all* values of variables.

Construct 6.7. *The family $\mathcal{C}_V$ of Definition 6.5 comprises the cocones of Construct 4.15 used in Definition 4.9 to generate the abstract state category of a Basic State system, as well as the cocone defined by the edges representing the `finext` and `ficurrent` functions, for any component `fi`.*

The family $\mathcal{C}_V$ represents those datatypes which are defined constructively, including the `fi-state` datatype for any component `fi`. This datatype was not represented in the cocones defined in Construct 4.7, which were used to generate the abstract state category of the underlying Basic State system. This is because the relevant functions (namely, `finext`) are not represented within the abstract state category of a Basic State system.

A model of the category $\mathbf{C}_V$ of Definition 6.5 is then a functor to **Set**, preserving finite limits and countable coproducts. Such a functor represents the current values of all variables (that is, a Basic state R), as well as one possible trace for each component in the system, originating at a Basic state R. A family of these functors can then be used to describe all possible behaviours originating at R, for any component. While this is what is required for a Failure Tolerance state, we must now consider how such behaviours might appear given that a Failure Tolerance system makes use of an interface hiding certain information.

Using interfaces to hide information

In general, an interface will hide certain data elements, often those used for implementation. In Section 5.3 we discussed the use of a view, or inclusion, functor K to identify those data elements of a system which comprise a particular subsystem. To model an interface, therefore, we will define a view functor A to be used with the Failure Tolerance abstract state category of Definition 6.5.

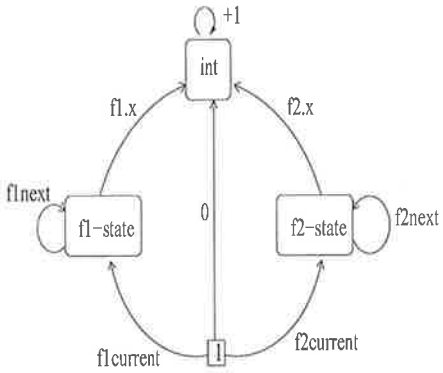

Figure 6.1: The category $\mathbf{C}_V^A$ represents current variables, as well as future ones.

This functor will identify the data elements visible under the interface. The data elements identified in this way form a subsystem which we will represent by a category termed the *abstract state equivalence category.*

Definition 6.8. *The* abstract state equivalence category $\boldsymbol{C}_V^A$ *of a Failure Tolerance system is defined in terms of the inclusion functor* $A : \boldsymbol{C}_V^A \rightarrow \boldsymbol{C}_V$*, where* $\boldsymbol{C}_V$ *is the Failure Tolerance abstract state category of Definition 6.5 and* A *identifies those elements of the system which are visible under the interface.*

Figure 6.1 shows the abstract state equivalence category for the Example 1 code of Section 4.3.1, assuming the designer has dictated that the variable y of component `f1` is hidden under the interface. We may usefully compare this with Figure 4.2, which represents the abstract state category for the same system interpreted under the Basic State definition.

Definition 6.9. *A model of the abstract state equivalence category is a functor* $R^A : \boldsymbol{C}_V^A \rightarrow \boldsymbol{Set}$ *which preserves finite limits and countable coproducts. This is also referred to as a* Failure Tolerance trace.

A Failure Tolerance trace represents a visible trace or behaviour of each component originating at a Basic state R and viewed using the interface A.

According to Definition 6.2, a Failure Tolerance *state* comprises the visible values of some Basic state R and all possible visible traces originating at R for each component. Thus, a Failure Tolerance state will consist of a collection of Failure Tolerance traces, all of which describe visible behaviours from the same Basic state R. We now introduce a formal definition of a Failure Tolerance state, which will allow us to consider whether two Basic states R, R' are equivalent under the interface. For this equivalence to be the case, the current values of visible variables in both R and R' must be identical, as well as the possible visible traces from each of these states. We formalise this below.

Equivalence of Basic states: defining a Failure Tolerance state

In a slight abuse of notation, we will denote a Basic state R viewed under the interface by a functor $R \circ A$. To formalise this, we need simply to define a view functor A on the abstract state category of the underlying Basic State system, as in Section 5.3. This view functor A identifies those system elements visible under the interface, which are precisely those elements — with the exception of `fi-current` and `fi-next` for any component `fi` — identified by the view functor A of Definition 6.8.

There are two reasons why there might be several different visible traces originating at $R \circ A$, where R is a Basic state, and A represents the interface. Firstly, there might be several different possible values of any variable hidden under the interface. This means that two distinct Basic states R and R' might appear the same under the interface, yet the possible behaviours from each of these states might not be visibly identical. In this case, these two Basic states R and R' should generate distinct Failure Tolerance states. Secondly, there might be non-determinism present in the system. In this case, even if the interface hides no information at all, there may be multiple 'next' states of a Basic state R.

In order to formally define a Failure Tolerance state and ensure that it describes consistent visible behaviours originating at some $R \circ A$, we first must develop notation for identifying the current values which are described by a Failure Tolerance trace (ie. the current values visible under the interface). This is achieved by associating a Failure Tolerance trace R^A with some $R \circ A$, where R is a Basic state.

Definition 6.10. *Given a Failure Tolerance trace R^A, the* current trace value *$R^A|$ is the functor $R \circ A$, where R is a Basic state such that*

- *$R \circ A(a) = R^A(a)$ for any object a representing a datatype visible under the interface.*
- *$R \circ A(f) = R^A(f)$ for any morphism $f : a_1 \to a_2$ representing a function or constant visible under the interface.*
- *$R \circ A(x_{fi}) = R^A(fi.x \circ ficurrent)$, where x_{fi} and $fi.x$ represent a variable x of component `fi` visible under the interface.*

That is, $R^A|$ represents the visible values of a Basic state in which the behaviour described by the Failure Tolerance trace R^A might originate. For a given Failure Tolerance trace R^A, there might be several different Basic states R for which $R^A| = R \circ A$, and these different Basic states might have different visible futures. Because of this potential confusion, in the following section we discuss what it means for a Failure Tolerance state to be consistent.

Definition 6.11. *A Failure Tolerance state is a family of Failure Tolerance traces*

$$\{R^A : \mathrm{C}_V^A \rightarrow \textbf{\textit{Set}}\ \}_i$$

where for any ${R^A}_i$ *and* ${R^A}_j$ *in this family,* $|{R^A}_i| = |{R^A}_j|$

Under this definition, a Failure Tolerance state is simply a collection of Failure Tolerance traces which share a current trace value. We will sometimes refer to the *current trace value* $|\{R^A\}_i|$ of a Failure Tolerance trace $\{R^A\}_i$. This denotes the current trace value of all $R^A \in \{R^A\}_i$. However, as we have mentioned briefly above, not every Failure Tolerance trace in such a Failure Tolerance state will necessarily describe a consistent behaviour originating at the same Basic state R. In the following section we discuss some consistency conditions which will enforce this for Failure Tolerance states.

6.2.4 Consistency of Failure Tolerance States

A consistent Failure Tolerance state will consist of a family of Failure Tolerance traces, where each represents a trace of each component, originating at some common Basic state and viewed under the interface. These traces must obey the consistency conditions we discuss below.

Component-wise Consistency

A component-wise consistent Failure Tolerance trace represents a trace for each component (as viewed under the interface) originating at some Basic state R, where every state in this trace obeys the system constraints. Since the system constraints for each component are enforced in the system category $\mathbf{C}$, a component-wise consistent Failure Tolerance trace $R^A : \mathbf{C}_V^A \rightarrow \mathbf{Set}$ must factor through $\mathbf{C}$. To ensure that this factorisation is performed correctly, we define the following *Failure Tolerance state identification functor.*

Definition 6.12. *The* Failure Tolerance state identification functor *is a functor* $\bar{R} : \boldsymbol{C}_V \rightarrow \boldsymbol{C}$*, where* $\boldsymbol{C}_V$ *is introduced in Definition 6.5, and* $\boldsymbol{C}$ *in Definition 6.4. The state identification functor is* valid *if the following properties hold.*

- $\bar{R}$ *maps any object (representing a datatype) of* $\boldsymbol{C}_V$ *to the object of* $\boldsymbol{C}$ *representing this same datatype*
- $\bar{R}$ *maps any morphism (representing a function, variable, constant or function application) in* $\boldsymbol{C}_V$ *to the morphism of* $\boldsymbol{C}$ *which represents the same function, variable, constant or function application.*
- *For each component* fi*,*

  $$\bar{R}(ficurrent) = finext^k \circ fi\text{-}init$$

  for some $k \geq 0$*, representing the number of state transitions component* fi *has undergone.*

A Failure Tolerance state identification functor fulfills the same function as the state identification functor of Definition 4.20 does for a Basic State system. However, the different composition of the Failure Tolerance abstract state category means that such a functor has some additional consequences in a Failure Tolerance system.

Remark 6.13. *A valid Failure Tolerance state identification functor acts as the identity upon the morphism* `finext` *for any component* `fi`.

Stuttering occurs outside the normal sequence of state transitions. That is, although an infinite number of stutters can occur at any point during a behaviour, this is not reflected by using different state identification functors $\bar{R}$ for each stutter. Instead, we simply do not represent the stutters. This will be important in Section 6.2.5, when we describe transitions which appear to be stutters under the interface.

We can use the Failure Tolerance state identification functor to describe how a Failure Tolerance trace R^A factors through the system category $\mathbf{C}$. This factorisation is used in the following definition of Failure Tolerance component-wise consistency.

Definition 6.14. *A component-wise consistent Failure Tolerance trace is a functor* $R^A : \boldsymbol{C}_V^A \rightarrow \boldsymbol{Set}$ *for which there exists a valid Failure Tolerance state identification functor* $\bar{R} : \boldsymbol{C}_V \rightarrow \boldsymbol{C}$ *and system model* $D : \boldsymbol{C} \rightarrow \boldsymbol{Set}$ *such that*

$$R^A = D \circ \bar{R} \circ A$$

where A *is the view functor of Definition 6.8 representing the interface.*

Owing to the action of $\bar{R}$ emphasised in Remark 6.13, a component-wise consistent Failure Tolerance trace describes a behaviour of `fi` (for any component `fi`) originating at some Basic state R, in which each state within this behaviour is component-wise consistent. The Basic state R in question is known as the *associated* Basic state of this Failure Tolerance trace and identified in the following way:

Definition 6.15. *The* associated *Basic state* R *of a component-wise consistent Failure Tolerance trace* $R^A = D \circ \bar{R} \circ A$ *is a Basic state* R *for which*

$$R = (D \circ \bar{R} \circ A^I)|$$

where A^I *is a view functor representing the trivial interface which hides no variables.*

That is, we have temporarily circumvented the presence of the interface in order to identify the associated Basic state R of a Failure Tolerance trace R^A, making use of the current trace value introduced in Definition 6.10.

Interaction-wise Consistency

Interaction-wise consistency was introduced in Definition 4.19, and ensures that shared information has the same value when viewed from any component. An interaction-wise consistent Failure Tolerance trace R^A is one which represents a system behaviour viewed under the interface and originating at some interaction-wise consistent Basic state R. However, an interaction-wise consistent Failure Tolerance trace does not require that each Basic state in this behaviour be interaction-wise consistent. This implies that an interaction-wise Basic state R where there are no possible state transitions from R leading to interaction-wise consistent Basic states can still be interpreted as an interaction-wise consistent Failure Tolerance state. A consequence of this choice of definition is that we can represent Basic states which lead to inconsistency within the formal framework provided by CCF. This will allow us to apply the taxonomy of inconsistencies of Section 4.7 to systems which may also be affected by system failure.

To enforce interaction-wise consistency, we will define the *interaction-wise consistency category* of a Failure Tolerance system.

Definition 6.16. *The* interaction consistency *category $\bar{\mathcal{C}}_V$ for a Failure Tolerance system is the category presented by the CCF sketch $(G_V, \mathcal{D}_V \bigcup \bar{\mathcal{D}}_V, \mathcal{L}_V, \mathcal{C}_V)$, where*

- *G_V is the directed graph of Clarification 6.6, used in Definition 6.5 to generate the abstract state category of this Failure Tolerance system*
- *$\mathcal{D}_V$ is the family of pairs of paths in G_V of Definition 6.5, representing the axioms of the data universe.*
- *$\bar{\mathcal{D}}_V$ is a family of pairs of paths in G_V representing constraints which enforce shared information to have a single value at any time*
- *$\mathcal{L}_V$ and $\mathcal{C}_V$ are the families of cones and cocones in G_V used in Definition 6.5 to generate the abstract state category of this Failure Tolerance system*

Clarification 6.17. *$\bar{\mathcal{D}}_V$ is a set of paths in G_V with common source and target, where each pair is of the form*

$$f1.x \circ f1current = f2.x \circ f2current$$

for a variable x visible to components `f1` and `f2`.

The interaction-wise consistency category can thus be seen to represent the constraints $\mathcal{D}_V$ of the data universe, as well as those constraints $\bar{\mathcal{D}}_V$ which are applied to shared information to ensure interaction-wise consistency. If a Failure Tolerance trace R^A can be factored through the interaction consistency category of Definition 6.16, it will represent behaviours from an interaction-wise consistent Basic state R.

Definition 6.18. *A Failure Tolerance trace* $R^A : \boldsymbol{C}_V^A \rightarrow \boldsymbol{Set}$ *is* interaction-wise consistent *when there exists a functor* $Y : \bar{\boldsymbol{C}}_V \rightarrow \boldsymbol{Set}$ *such that*

$$R^A = Y \circ I \circ A$$

where $I : \boldsymbol{C}_V \rightarrow \bar{\boldsymbol{C}}_V$ *is the quotient functor.*

Complete Consistency of a Failure Tolerance State

A consistent Failure Tolerance state $\{R^A\}_i$ will be a family of Failure Tolerance traces, where each describes a behaviour of any component originating at a component-wise and interaction-wise consistent Basic state R, and viewed under the interface. Moreover, the behaviour described by any member of this family $\{R^A\}_i$ for a component `fi` consists of component-wise consistent states of `fi`. We formalise this by the following definition.

Definition 6.19. *A consistent Failure Tolerance state* $\{R^A\}_i$ *is one for which for each* $R^A \in \{R^A\}_i$

- R^A *preserves finite limits and countable coproducts*
- R^A *is a component-wise consistent Failure Tolerance trace*
- R^A *is an interaction-wise consistent Failure Tolerance trace*
- *Each* R^A *is* associated *(Definition 6.15) with the same Basic state* R

Consistency of a Failure Tolerance state is equivalent to the requirement that the diagram in Figure 6.2 commute for each Failure Tolerance trace within the family $\{R^A\}_i$. Thus, a Failure Tolerance state represents a family of all possible visible behaviours for each component, from some Basic state R. By extension, this is a representation of all possible visible behaviours of the system, both consistent and inconsistent, from R. Whether or not the system behaviour represented by a Failure Tolerance trace is consistent now depends only upon the scheduling of individual component state-transitions — that is, upon the interaction-wise consistency of future states.

6.2.5 The category of Failure Tolerance States

We now define the category of Failure Tolerance states, given an interface modelled by a view functor A as introduced in Definition 6.8. This category will be referred to as **Failure Tolerance(A)**. Objects in this category will be consistent Failure Tolerance states, while morphisms will represent behaviours of the Failure Tolerance system.

Definition 6.20. *The category* ***Failure Tolerance(A)*** *is comprised of*

- *objects which are consistent Failure Tolerance states as presented in Definition 6.19*

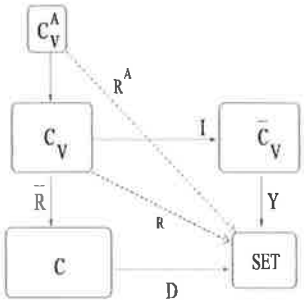

Figure 6.2: This diagram, in which all categories are Failure Tolerance categories, commutes if the Failure Tolerance sequence R^A is consistent.

- *morphisms which are of two classes:*
  1. *Operational morphisms, which are tuples* $r = (\{R^A\}_j, [R^A]_i, \{R^{A'}\}_k)$ *of current trace values of objects in* ***Failure Tolerance(A)****, satisfying Property 6.25 (described below as the Failure Tolerance State Ordering Property)*
  2. *Hidden morphisms, which are tuples* $r = (\{R^A\}_j, [], \{R^{A'}\}_k)$ *where*

$$\{R^A\}_j| = \{R^{A'}\}_k|$$

In this definition, operational morphisms represent consistent behaviours of the system, as distinguished by the visible values observed throughout each behaviour. Hidden morphisms represent the behaviour of a system when state is restored after system failure. A morphism may be both operational and hidden, and we provide clarifications and discussions of both types of morphisms in the rest of this section.

Objects in the Failure Tolerance Category

Defining equality of objects in **Failure Tolerance(A)** can be relatively complex. Informally, two Failure Tolerance states should be deemed equal as objects in **Failure Tolerance(A)** when they are associated with *equivalent* Basic states R and R'. However, for the sake of completeness, it should not be necessary to analyse the underlying Basic State system to identify equal **Failure Tolerance(A)** objects. That is, the categorical representation of a Failure Tolerance system should not rely on the presence of a categorical representation of the underlying Basic State system. Instead, we require that equality of two Failure Tolerance states be equality of the Failure Tolerance traces of which they are composed, modulo any stuttering. This leads to the following definition of equality of Failure Tolerance states, which we then discuss in more detail.

Definition 6.21. *Two objects* $\{R^A\}_i$ *and* $\{R^{A'}\}_i$ *of* ***Failure Tolerance(A)*** *are equal if for every Failure Tolerance sequence* R^A *in* $\{R^A\}_i$ *there is a Failure Tolerance trace* $R^{A'}$ *in* $\{R^{A'}\}_i$ *such that*

$$R^A = R^{A'}$$

up to the conflation of consecutive identical visible states of any component `fi` *in the behaviour of* `fi` *described by either Failure Tolerance sequence.*

This ensures that two Basic states R and R' which are equivalent under the interface A generate equal states in the **Failure Tolerance(A)** category. That is, if Basic state R and R' are equivalent under A, then the possible behaviours described for each component from either state must be visibly identical. This means the Failure Tolerance traces which make up the Failure Tolerance states associated with Basic states R and R' must be identical. Definition 6.21 also ensures that if two Basic states appear identical under the interface, yet result in visibly different system behaviours, these Basic states are not associated with the same object in the **Failure Tolerance(A)** category.

However, as we stated briefly earlier, any analysis we perform is to be stuttering-insensitive. This characteristic has been reflected in the categories **States**, **TH States** and **Input States** by the definitions of morphisms. In the category **Failure Tolerance(A)**, however, we must consider the implications of stuttering when defining equality of objects. A Failure Tolerance sequence is constructed by assuming that no stutters are present — an abstraction which can be safely made since stuttering is intended to be irrelevant. However, a Failure Tolerance sequence may describe a state transition of some component which is not a stutter, yet appears to be one when viewed under the interface. These are known as *false stutters*, and any analysis we perform should be insensitive to these false stutters as well as 'genuine' stutters. Definition 6.21 ensures that this is the case, by conflating visibly identical states in any behaviour.

Definition 6.22. *A* false stutter *of a component* `fi` *is a state transition of* `fi` *which is not a stutter, but appears to be so when an interface hides some of the system data of* `fi`.

Definition 6.21 provides criteria for two Failure Tolerance states to be equal, given the 'meta-knowledge' that in all applications a Failure Tolerance trace R^A defines a visible behaviour for each component. However, a formal definition of equality in the category **Failure Tolerance(A)** should not rely on this practical information about the use of Failure Tolerance traces. With this in mind, we present the following detailed discussion of equality of objects in **Failure Tolerance(A)**. This is simply a formalisation of the concepts presented in Definition 6.21, and familiarity with the following property (Property 6.23) and definition (Definition 6.24) is not required in order to use the analysis techniques of this chapter.

A Formal Treatment of Equality

In order to establish a formal definition of equality of objects in **Failure Tolerance(A)**, we introduce the following property. This will hold for any category of Failure Tolerance states.

Property 6.23 (The Hidden Transition Property). *For any object $\{R^A\}_j$ of the category **Failure Tolerance(A)**, for all Failure Tolerance sequences $R^A \in \{R^A\}_j$ and for any integer i, there exists some natural number n (which may be 0) such that for each component `f1` and for all morphisms `f1.x`, where `f1.x` represents a variable x in `f1`*

$$R^A(f1.x \circ f1next^{i+r} \circ f1current) = R^A(f1.x \circ f1next^{i} \circ f1current)$$

for all $r \leq n$.

This property states that any Failure Tolerance trace R^A describes a behaviour for any component `f1` originating at the Basic state R associated with R^A, where after i transitions into the behaviour there are n false stutters of `f1`. We note that Property 6.23 for all Failure Tolerance traces R^A including those which describe no false stutters at all, since n may be 0.

We then define a function $\mathrm{Run}_{f1}^{R^A}(i)$, where $\mathrm{Run}_{f1}^{R^A}(i)$ uses Property 6.23 to obtain the length n of the sequence of false stutters for `f1` *beginning* at the i-th state transition of `f1`, as described by the Failure Tolerance sequence R^A. This function is defined as follows:

$$\mathrm{Run}_{f1}^{R^A}(i) = 0 \text{ if } \mathrm{Run}_{f1}^{R^A}(i-1) \neq 0$$
$$\mathrm{Run}_{f1}^{R^A}(i) = n \text{ otherwise.}$$

That is, $\mathrm{Run}_{f1}^{R^A}(i)$ provides the length of the sequence of false stutters of `f1` which *begins* at the i-th transition of `f1` described by R^A. If the i-th transition is itself in the middle of such a run then $\mathrm{Run}_{f1}^{R^A}(i)$ is 0.

Since the Hidden Transition Property holds for all Failure Tolerance traces R^A in every Failure Tolerance state $\{R^A\}_i$, we may use this to define precisely when two objects in **Failure Tolerance(A)** are equal, modulo stuttering.

Definition 6.24. *Two objects $\{R^A\}_i$ and $\{R^{A'}\}_i$ of **Failure Tolerance(A)** will be equal when for any functor $R^A \in \{R^A\}_i$ there is a functor $R^{A'} \in \{R^{A'}\}_i$ (and vice versa) such that*

- *R^A and $R^{A'}$ act identically upon any object in $\boldsymbol{C}_V^A$, or any morphism in $\boldsymbol{C}_V^A$ representing a function or constant.*
- *For all morphisms $f1.x$ in $\boldsymbol{C}_V^A$ representing a variable x of any component `f1`, then for any integer k*

$$R^A(f1.x \circ f1next^{k+\sum_{r=0}^{k}(\mathrm{Run}_{f1}^{R^A}(r))} \circ f1current) =$$
$$R^{A'}(f1.x \circ f1next^{k+\sum_{j=0}^{k}(\mathrm{Run}_{f1}^{R^{A'}}(j))} \circ f1current)$$

The last condition of Definition 6.24 is a formal means of defining equality up to conflation of any consecutive identical states of any component, as defined by a Failure Tolerance sequence. As has been the practice in previous categories, all functors are defined up to natural isomorphism only.

This definition is presented for the sake of completeness only, and to establish the categorical foundation of this particular aspect of CCF. In most cases, Definition 6.21 is sufficiently precise.

Morphisms in the Failure Tolerance Category

Morphisms in the category **Failure Tolerance(A)** are divided into two classes. The first class, the *operational morphisms*, are those morphisms which represent consistent behaviours of the Failure Tolerance system. In accordance with Definition 6.2, behaviours of a Failure Tolerance system are distinguished by the visible values of variables throughout the behaviour. Thus, these *operational* morphisms are tuples

$$r = (\{R^A\}_j, [R^A|]_i, \{R^{A'}\}_k)$$

defining the current trace values which are visible throughout each behaviour. However, not all possible tuples will represent a consistent behaviour. In order for a tuple to represent a consistent behaviour, the system constraints relating one state to another must be satisfied. We apply this criteria to a tuple r by means of the following property.

Property 6.25 (The Failure Tolerance State Ordering Property). *A tuple* $r = (\{R^A\}_j, [R^A|]_i, \{R^{A'}\}_k)$ *satisfies the Failure Tolerance State Ordering Property if the following conditions hold:*

- *for all Failure Tolerance sequences* ${R^A}_j$ *and* ${R^A}_{j+1}$ *where* ${R^A}_j|$ *and* ${R^A}_{j+1}|$ *are consecutive current trace values in* r*, and where*

$$ {R^A}_j = D \circ \bar{R}_j \text{ and} $$
$$ {R^A}_{j+1} = D \circ \bar{R}_{j+1} $$

  for some system model D *and Failure Tolerance state identification functors* $\bar{R}_j$ *and* $\bar{R}_{j+1}$*, the following equalities hold for any component* `fi`*:*

  1. $\bar{R}_{j+1}(ficurrent) = \bar{R}_j(finext \circ ficurrent)$ *or*
  2. $\bar{R}_{j+1}(ficurrent) = \bar{R}_j(ficurrent)$

A tuple r satisfying this property is a tuple of Failure Tolerance sequences where the current trace values of these sequences describe a consistent behaviour of the system. This is precisely the property required of an operational morphism.

Clarification 6.26. *The* operational morphisms *in* ***Failure Tolerance(A)****, introduced in Definition 6.20, are tuples* $r = (\{R^A\}_j, [R^A]|_i, \{R^{A'}\}_k)$*, where*

- $\{R^A\}_j$ *and* $\{R^{A'}\}_k$ *are objects in* ***Failure Tolerance(A)***
- $[R^A]|_i$ *is a list of current trace values of objects in* ***Failure Tolerance(A)***
- r *satisfies Property 6.25 (the Failure Tolerance State Ordering Property)*

Each operational morphism r then describes a consistent behaviour of the **Failure Tolerance(A)** category. That is, unlike the categories **States**, **Input States** and **TH States**, we incorporate the definition of the subcategories **Failure Tolerance(A)(D)** (consisting of those morphisms which are defined over the system model D) into the the state ordering property (Property 6.25). This is because in **Failure Tolerance(A)**, future Basic states of each component are part of the definition of a Failure Tolerance state. Thus, it makes no sense to allow morphisms which are not defined over a single D, as these will not be represented within any Failure Tolerance state. Equality of operational morphisms is equality of the tuples, up to the conflation of consecutive identical elements (current trace values) of each tuple, and composition is concatenation.

As we discussed in Section 6.1, we will use this category **Failure Tolerance(A)** to examine the effect of restoring the visible system state after a system failure. To facilitate this we defined another class of morphisms in **Failure Tolerance(A)**, where these morphisms represent the system behaviour which corresponds to the restoration of visible system state.

Clarification 6.27. *The* hidden *morphism in* ***Failure Tolerance(A)****, introduced in Definition 6.20, are tuples* $r = (\{R^A\}_j, [], \{R^{A'}\}_k)$ *where*

$\{R^A\}_j$ *and* $\{R^{A'}\}_k$ *are objects in* ***Failure Tolerance(A)*** *such that*

$$\{R^A\}_j| = \{R^{A'}\}_k|$$

A hidden morphism can be used to describe the visibly correct restoration of a Basic state R which generates the Failure Tolerance state $\{R^A\}_j$. The visible values are guaranteed to be restored correctly, a fact given by the equality

$$\{R^A\}_j| = \{R^{A'}\}_k|$$

but the hidden values may be restored incorrectly, leading to the situation where

$$\{R^A\}_i \neq \{R^{A'}\}_k.$$

That is, where the Failure Tolerance sequences making up $\{R^A\}_i$ differ from the Failure Tolerance sequences making up $\{R^{A'}\}_j$. There is precisely one hidden morphism for any two such objects $\{R^A\}_i$ and $\{R^{A'}\}_j$ where $\{R^A\}_j| = \{R^{A'}\}_k|$. The composition of two morphisms $r : \{R^A\}_i \to \{R^{A'}\}_j$ and $r' : \{R^A\}'_j \to \{R^A\}_i$ is the identity morphism.

Note that it is possible for a morphism to be both operational and hidden. For example, the identity morphism belongs to both of these classes. To enhance clarity, we will reserve the term *restoration morphisms* to describe the actions specifically corresponding to restoring visible system state, rather than observing a consistent behaviour. Where we need a term for this class in general, they shall be referred to as *hidden* morphisms.

6.2.6 Applying the Failure Tolerance(A) category

The **Failure Tolerance(A)** category is obtained by defining an interface A to be used with a Basic State system. Each Basic state of the underlying system is then associated with a Failure Tolerance state, which describes the current visible variables, and visible behaviours originating from this Basic state. Two equivalent Basic states will then generate equal Failure Tolerance states. Definition 6.19 formally describes states as objects in the category **Failure Tolerance(A)**, while Definitions 6.26 and 6.27 describe behaviours as morphisms. Using **Failure Tolerance(A)** allows us to analyse the observable behaviours of a system during system failure, given that variables hidden under A cannot be guaranteed to be restored correctly. In the following section, we will show how this can be used in a real-world application.

6.3 System Failure in Specifications Utilising an Interface

In Section 6.1.1 we introduced the concept of an interface hiding certain information within the system. The information visible under such an interface can be restored accurately after system failure, as it has been independently observed by another external entity. When restoring state after a system failure, a user is primarily concerned about restoring a state which appears identical to the one in which failure occurred. Such a restoration, however, may not always result in completely consistent behaviour.

In Section 6.3.1 we present an example of a system in which an interface is present. This is a security system which is specified at a relatively abstract level, and which will serve as a case study for some discussions of failure tolerance. Failure tolerance was introduced in Section 6.1.1, and refers to the ability of a system to recover from system failure. In many cases, it is too ambitious to require an entire system to recover completely from such a failure. Instead, a designer might identify essential subsystems, and specify that the degree of inconsistency these display in the presence of system failure be limited. In the rest of this chapter, we discuss how the overall design of the system can allow for the identification of essential subsystems, and the requirement that they display a minimum degree of consistency at all times.

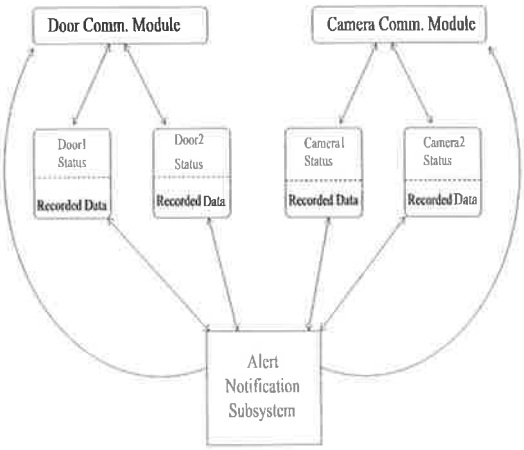

Figure 6.3: A security system, including both visible and hidden data.

To facilitate this discussion, we will use the category **Failure Tolerance(A)** of Section 6.2.5 to provide a formal treatment of the security system case study. Within this context, we then identify a subsystem which is deemed "essential" to the correct operation of this security system. We can then use **Failure Tolerance(A)** to express the relationship between behaviours of this essential subsystem and behaviours of the wider system. Section 6.4 introduces some mathematical constructions which we will use to limit the inconsistency of this essential subsystem in the event of system failure. This is known as ensuring *conditional failure tolerance* of the subsystem. Finally, in Sections 6.5 and 6.7 we discuss two fundamental behaviours of the security system, and show how CCF can be used to enforce conditional failure tolerance of the essential subsystem with respect to these behaviours.

6.3.1 A Security System Example

The example we will be using to illustrate conditional failure tolerance is the specification of a security system. Many of the characteristics of this security system are taken from the features offered by Chubb [16] for monitoring high-risk buildings, and part of the physical composition of the system is depicted in Figure 6.3.

Security Sensors

This system will consist of several sensors or modules, each of which is represented by a component in the specification. The sensors monitor the building in which the security system is installed, and make some of this information visible to a user. For example, each door is represented by a sensor, for which

the status may be `open` or `closed`. This status is always visible to the user of the security system. In addition to the status, a sensor may provide additional visible information to the user of the system. For example, a door sensor might display a message if a code is needed to open the door. For cameras, the information visible to the user of the security system corresponds to the status of the camera (for example, whether it is operational or not, and what mode of operation it is currently in), and the image shown at that moment.

A door communication module establishes communication between the doors, handling any issues of coordination as well as basic shutdown and maintenance procedures. This structure is repeated for the other observational sensors, such as cameras. For example, the camera communications module acts as a sequencer, bypassing any input from unused camera units and automatically switching monitors. In addition to these door and camera modules there may be other components of a security system, such as a number of guard personnel tracked by alarm locator units.

In addition to the information visible to the user, each sensor will typically retain a record of events that have involved the component in question. For example, a camera sensor typically preserves a record of those segments of video where the recording was triggered by a motion sensor. A door sensor, on the other hand, might preserve a record of the codes which have been used to open this door in the current monitoring session. These records should not be available to a user of the security system, instead being uploaded to an external repository upon system shutdown. This reason for this restriction is because a user of the security system — who will usually not be a qualified security technician — should only see what is transpiring in the building at the current time. Allowing him to see (and therefore potentially alter) the recorded data results in additional complexity of the system, or even corruption of the recordings.

The alert notification subsystem

In order for this system to function as more than a disparate collection of separate sensors, we need a central alert notification component. The job of such a component is two-fold. Firstly, it must coordinate any communication the system has with the external environment. This includes sending an alarm to the police, or requesting that an external monitoring service be alerted. Secondly, it must coordinate actions of one sensor with the other sensors which make up the security system. Specifically, the alert notification component interprets information from one sensor and tells the others how they should react to the situation. For example, a change in the status of the sensor representing a security guard's callbutton may trigger a notification for outer doors to lock down.

In order to make the implementation of the security policies transparent to a user, all the data which is part of the alert notification component is made visible. This means the user always knows the state of this notification component, and is therefore aware of the messages being transmitted within the security system at any given time.

Security Policies: Stating Required Behaviours

The behaviour of the security system is constrained by a number of Security Policies. Those Policies that we present here demonstrate the type of requirements which are obtained very early on in the development cycle. By using these, we will show that ensuring failure tolerance can be done throughout the software lifecycle, even when the requirements and specifications are still incomplete. Additionally, using incomplete requirements will allow us to demonstrate how refining the system can affect failure tolerance. Finally, we will also show how ambiguities stemming from informal requirements documention may be resolved.

In general, we require that the alert notification subsystem finish sending any notification before it can send the next. This ensures consistency, in that we know that all the actions taken in response to the first notification are completed before the next is sent. Finally, as a general principle, we require that the recorded data cannot be erased by any notification. This ensures that a malfunctioning system which sends incorrect notifications cannot erase a record of exactly when the malfunction occurred. There are some simple Security Policies we can write which encapsulate these characteristics:

```
                    Security Policies

1) Pre-condition: Any previous notifications must have been
                  completed and acknowledged.
Action: The alert notification subsystem sends out a notification.
Post-condition: The recorded data which existed before this
                notification must still exist.
```

At the early stages of development — which is where we assume this case study lies — the specification for the alert notification component should be as generic as possible. This facilitates code re-use and enables us to easily refine the specification. To this end we define certain *classes* of notification. Each class corresponds to a family of notifications which all display certain properties. For example, one common way in which to categorise notifications is into the following families:

1. Calls to an external source (eg. to the company which supplies the system, or to the police).

2. Notifications broadcast to the entire system.

3. Notifications sent to an individual sensor.

This categorisation is based upon the different actions required for the alert notification system to send messages of each class. For example, broadcast messages can be sent only under certain circumstances, which differ from the circumstances under which an external call would be made. We discuss this further when explaining Security Policy 2. At this early stage of development, we will consider the circumstances under which messages of each class would be sent, rather than examining individual messages.

The first class of messages we examine are the broadcast messages. These are typically low-priority maintenance messages, as in a high-risk situation the alert notification will communicate individually with each sensor to ensure that the correct outcome is achieved. When broadcasting messages, however, the alert notification component will communicate only with the sensor communication modules, such as the door and camera communication modules of Figure 6.3. By interpreting broadcast messages as low-priority, we can create a hierarchy of notification classes. This hierarchy allows the system to prioritize certain alerts, or allow others to be sent only under certain highly-constrained circumstances. For example, in a high-alert state the system may refuse to send any broadcast messages. This ensures that there will never be a conflict between responding to a high-priority alert and sending a routine maintenance message.

This prioritization method requires the system to identify high-alert states. The identification criteria will be different for each system, and will typically be customized by the user. The action taken in one of these high-alert states is also dependent upon the system and the preferences of the user. In this security system example, the recommended action in such a state is to send an external call — to the police, or the security monitoring company. The following Security Policy encapsulates this decision for this system.

```
                    Security Policies

2) Pre-condition: An alarm of sufficient importance is sent from a
                  sensor module to the alert notification system.
Action: The system enters a 'high-alert' state.
Action: The alert notification system sends out an external
        alarm call.
Post-condition: The system remains in a 'high-alert' state until
                the external call has been answered.
```

If the system is in a high-alert state then it may refuse to carry out any expected routine maintenance. The reason for this refusal should be communicated to the user, who might otherwise suspect the system of malfunctioning. This motivates the following additional post-conditions for this Security Policy.

```
Post-condition: The fact that the system is in a high-alert state
                is visible to the user.
Post-condition: Broadcast messages will not be sent until the
                system returns to a normal state.
```

In general, the user of the security system will be able to tell whether an alert is being sent as a broadcast message or a series of messages to individual sensors. This is because the operation of the alert notification subsystem is completely transparent to any user, and the actions of this component required to send the two types of alerts differ. This means that any violation of Security Policy 2 can be detected by a user.

Finally, a security system will usually include some safety policies which are typically added relatively late in the design cycle. The following policy enforces a safety consideration, that when an emergency personnel member has opened a door, he must sign out (verify his identity) before this door can be closed. This security policy will be used only in Section 6.6, in which we show how adding Security Policies can alter the failure tolerance of a system. That is, for all analysis before Section 6.6, Security Policy 3 does not apply.

Security Policies

```
3) Pre-condition: The code last used to open a door was an emergency
                  personnel code.
Action: A request to close the door is sent without an emergency
        personnel closure code.
Response: The request is denied.
Post-condition: The door remains open.
```

Note that this does not gurantee that the door *will* be closed if a request is sent and the code has been entered.

In the following two subsections we present two fundamental behaviours of the security system as they might be described at an early stage of specification. The first describes the actions that should be taken in response to a general alert.

Behaviours of the Security System: Responding To An Alert

When specifying a security system, one of the most fundamental considerations is what actions the system should take in response to an alert. While the specific actions depend, naturally, upon the alert in question, at this early stage of the design it is also worth describing a general response. In the absence of such a description, for example, a module might non-deterministically choose whether or not to act on the alert. To avoid this ambiguity, we present the following Security Policies:

```
4) Pre-condition: Module A can achieve outcome X.
Precondition: An event occurs causing the alert notification
                system to request outcome X from module A.
Action: An alert is sent to module A requesting outcome X.
Action: Module A takes actions that will result in X and sends
          an acknowledgement and confirmation to the alert
          notification subsystem.
Requirement: The actions taken to achieve X must be predictable.

5) Pre-condition: Module A cannot achieve outcome X.
Precondition: An event occurs causing the alert notification
                system to request outcome X from module A.
Action: An alert is sent to module A requesting outcome X.
Action: Module A takes no action and sends an acknowledgement
          and denial to the alert notification subsystem.
Post-condition: No recorded data in the system is altered by this
                communication
```

For example, if the door communications module receives an alert requesting that a certain door change status from `shut` to `open`, the door communications module must first perform this action — sending the request to the individual door sensor — then acknowledge that this outcome was achieved. Security Policy 4 ensures that a successful response to an alert will always *include* the desired outcome, yet recognises that in some cases there is no need for this response to be *restricted* to this outcome. The requirement that the actions taken to achieve outcome X be 'predictable' is deliberately vague at this level of specification. It is intended to ensure a designer implements a particular algorithm for achieving each desired outcome rather than leaving the action unspecified. However, it does not constrain him to implement any particular one of the possible algorithms.

It may also be the case that for whatever reason, the module in question cannot achieve the outcome requested in the alert. In this case, Security Policy 5 dictates that no action may be taken, beyond acknowledgement that the notification was received. This will become an issue both in terms of the frame problem — how should such a 'non-action' be expressed and guaranteed? — and in terms of refinement of the system. In general, because the suggested response to an alert can be phrased only in very general terms, it is one of the system properties most likely to change with subsequent refinements, or suffer from interactions with the pre-conditions of new security policies.

Behaviours of the Security System: Pre-set Security Levels

The second behaviour we present describes the way in which a user can interact with a security system — namely, by selecting the desired level of security. Different security levels enable particular families of sensors to be activated, while others are only partially activated or disabled entirely. Providing some pre-set security levels means that it is easier for a user to change the most common security settings.

At this stage of specification, we do not suggest an implementation for sending the requests to change security settings out to the system. Depending upon the customer's requirements, the change of security settings may be considered a low-priority message and sent via broadcast, or alternatively may be considered high-priority and sent individually to all affected sensors. These different ways of dealing with user input are both valid implementations, and can be considered as opposite ends of a prioritization spectrum, with the potential for the customer to require an implementation which is anywhere along this spectrum. The final consideration when discussing pre-set security levels is that the provision, explanation and presentation of these security settings must be user-friendly [48] and not rely on a knowledge of the system. That is, a user without technical knowledge about this specification should be able to interact with this security system. The problem with this informality, as we will demonstrate in Section 6.5, is that it can lead to inconsistencies between a user's expectations and the constraints of detailed Security Policies.

Common pre-set levels of security, as they might be presented to an end-user, are:

1. **No Security:** All doors are opened, metal detectors are off, cameras are off and the security guard alarms are disabled. This can be achieved simply by turning the system off.
2. **Minimum Security:** Inner doors are shut and require a code to enter, cameras are off, the security guard alarms are disabled, outer doors are open, and all metal detectors are off.
3. **Moderate Security:** All inner doors are shut and require a code to enter, cameras are switched on, the motion detection part of the camera system is enabled for the inner areas only, and metal detectors are switched on.
4. **Maximum Security:** All doors are shut and require a code to enter, all cameras and motion sensors are switched on, metal dectectors are activated, and the security guard alarm is activated in every room.

6.4 CCF, System Failure and Degrees of Inconsistency

It is fundamental to the correct operation of any system that its behaviour is constrained by the system specification. In the security system example of Section 6.3.1, this means that all behaviours must be constrained by the Security Policies. In theory we also wish to ensure that a system failure does not result in a behaviour which violates the user's expectations in any way. However, as we have mentioned before, complete failure tolerance like this is a very strict condition and may be impossible to achieve. As a compromise between theory and practice, we instead identify essential subsystems of the wider system. We will require these essential subsystems to display *at least* a minimum degree of consistency at all times. In the process of expressing this formally, we will make use of techniques from Section 5.3 relating to subsystem behaviour.

We will be using the security system of Section 6.3.1 to illustrate failure tolerance of essential subsystems. Here, we identify the alert notification subsystem as an essential component of the security system. Instead of requiring that the entire system recover completely from system failure, we will instead ensure that a system failure cannot result in the alert notification subsystem somehow communicating 'incorrectly' with the sensors.

As we have mentioned earlier, the notion of system failure is somewhat nebulous and its effects hard to quantify. To reduce the ambiguity associated with specifying how a system recovers from failure, we use some of the formalisms of CCF to model the recovery of a system. Specifically, we will make use of the **Failure Tolerance(A)** category, introduced in Section 6.2, and the minimal behaviours discussed in Section 5.3. With the help of these categorical constructions, we will show how to formally define the degrees of failure tolerance of a subsystem.

6.4.1 Failure Tolerance in the Security System

The security system of Section 6.3.1 makes certain information visible to an observer, and therefore guarantees that it will be restored correctly after system failure. The information which is *not* made visible to an observer, such as the recorded data of each sensor, cannot be guaranteed to be restored correctly. By contrast, the current status of any sensor (information which *is* made visible to an observer) can be restored accurately after power failure. For example, the status of a door can be restored simply by using a physical detector to see if the door is open or closed. Likewise, a camera will retain the current image at the time the power went off within a built-in memory [16]

As we discussed earlier, all the information within the alert notification subsystem will be made visible to the user, often by a visible display summarising

the state of all messages in progress. One consequence of making this information visible this is that any restoration after system failure will be accurate upon the information within the alert notification system. However, because the actions of the alert notification subsystem are governed in part by the wider system, this visibility of information does not guarantee that the alert notification subsystem will be completely failure tolerant. That is, if the restoration is inaccurate upon the rest of the system, then this may lead to inconsistent behaviour of the alert notification component.

In order to discuss the behaviour of this security system in the presence of system failure, we will model it as a Failure Tolerance system. The category of states will be denoted **Failure Tolerance(A)**. The view functor A identifies the information which is visible to a user of the security system, and which is therefore able to be restored correctly. Because the alert notification component does not include any hidden variables, it can be interpreted as a Basic State system. The category of states of the alert notification subsystem will be denoted $\mathbf{States}_{ANC}$ and generated by the techniques described in Chapter 4. Moreover, since the alert notification component is a subsystem of the overall security system, we can define a view functor K which identifies the system elements which make up the alert notification subsystem. This type of view functor was introduced in Section 5.3, and an example can be seen in Figure 5.5. The functor K^*, representing composition with K, relates states and behaviours of the security system to the relevant states and behaviours of the alert notification subsystem.

Definition 6.28. *The category of states of the alert notification subsystem, interpreted as a Basic State system, is denoted* $\boldsymbol{States}_{ANC}$. *A functor*

$$K^* : \boldsymbol{Failure\ Tolerance(A)} \rightarrow \boldsymbol{States}_{ANC}$$

relates states and behaviours of the underlying security system to states and behaviours of the alert notification subsystem.

We emphasise here that we use two different view functors when analysing the security system with CCF. The first, A, represents the interface and identifies those variables which are visible to an observer — for example, the current value of any sensor. The second, K, identifies the alert notification subsystem. Every variable identified by K is also identified by A, but the converse is not necessarily true. We also note that use of K^* requires that we treat the alert notification subsystem as a Failure Tolerance system, in order to be categorically well-defined. This is not a problem, as according to Remark 6.3 any Basic State system can be interpreted as a Failure Tolerance system with a trivial interface.

Since the alert notification subsystem is a Basic State system, when removed from its environment the values in its next state are determined solely by the values in its current state (modulo non-determinism). However, when placed

in the environment of the wider system the behaviour of the alert notification subsystem may be dependent upon the values of variables within the wider system. This can include variables, or information, which is not visible to an observer. That is, given two objects $\{R^A\}_i$ and $\{R^{A'}\}_i$ of the category **Failure Tolerance(A)** of states of the security system for which the current trace values are the same

$$\{R^A\}_i| = \{R^{A'}\}_i|$$

then these states $\{R^A\}_i$ and $\{R^{A'}\}_i$ will restrict to the same state T of the alert notification subsystem. However, the families $\{R^A\}_i$ and $\{R^{A'}\}_i$ may not describe the same behaviours of the subsystem. Additionally, there may of course be other system states $\{R^{A''}\}_i$ for which $\{R^{A''}\}_i|$ restricts to subsystem state T, but for which

$$\{R^{A''}\}_i| \neq \{R^A\}_i|$$

For ease of notation, we will henceforth refer to a Failure Tolerance state $\{R^A\}_i$ simply as R. Since we will never discuss the underlying Basic states of the security system, this should cause no confusion. Two Failure Tolerance states will be referred to as *visibly identical* if their current trace values are the same, and two Failure Tolerance morphisms as *visibly identical* if they describe the same current trace values.

Definition 6.29. *Two morphisms* $r = (\{R^A\}_j[R^A|]_m\{R^{A'}\}_k)$ *and* $r_2 = (\{\mathrm{R}_2^A\}_j[R^A|]_n\{\mathrm{R}_2^{A'}\}_k)$ *in* ***Failure Tolerance(A)*** *are* visibly identical *if*

$$\begin{gathered}\{R^A\}_j| = \{\mathrm{R}_2^A\}_j| \\ \{R^{A'}\}_k| = \{\mathrm{R}_2^{A'}\}_k| \text{ and} \\ [R^A|]_m = [R^A|]_n \text{ as lists}\end{gathered}$$

Two objects in ***Failure Tolerance(A)*** *are* visibly identical *if they have the same current trace values.*

We will also refer to the *visible behaviour* indicated by a Failure Tolerance morphism $r = (\{R^A\}_j[R^A|]_m\{R^{A'}\}_k)$. By this, we mean the sequence $[R^A|]_m$ of current trace values represented by this morphism.

We introduce this notation to emphasise the importance of the *visible* properties of a system state or behaviour, especially with respect to any essential subsystems. As we will see later, failure tolerance is dependent upon the perspective of an observer. Thus, if we require the alert notification component to demonstrate failure tolerance, it is sufficient for there to be no visible difference between its behaviours in the presence of, and in the absence of, system failure.

6.4.2 Treating Failure Tolerance Categorically

When using the category **Failure Tolerance(A)** to analyse the effect of system failure, we are interested in the interaction of operational and restoration

morphisms. In particular, we are interested in the situation where commutativity exists between these morphisms. That is, suppose $m_1 : R_1 \rightarrow R_2$ and $m_2 : {R_1}' \rightarrow {R_2}'$ are restoration morphisms, and $r_1 : R_1 \rightarrow {R_1}'$ and $r_2 : R_2 \rightarrow {R_2}'$ are operational morphisms. The commutativity $m_2 \circ r_1 = r_2 \circ m_1$, as shown in Figure 6.4, allows us to deduce the adequacy of the restoration m_1.

Specifically, if a commutative diagram such as that of Figure 6.4 can be obtained for an operational morphism r_1 and all restoration morphisms m_1, this implies that the ability to observe the visible behaviour denoted by r_1 is not affected by system failure. This is a form of *conditional failure tolerance*, since it implies that the potential for this behaviour to occur appears — to the user — unaffected by the failure. Informally, we might say the system is conditionally failure tolerant on those behaviours r_1 for which any restoration morphism m_1 results in the commutative diagram of Figure 6.4.

We can use this informal demonstration of conditional failure tolerance to motivate the formal definitions we present later of failure tolerance of essential subsystems. When discussing these subsystems, we will require that observing the behaviour of a subsystem P and knowing its degree of failure tolerance should allow us to deduce information about the behaviour of the system. This practice of deducing characteristics of the system from characteristics of a subsystem was first introduced in Section 5.3. To apply it within this context, in the following section we introduce the category of consistent system behaviours which restrict to a given subsystem behaviour t.

6.4.3 The category $\mathcal{E}(t)$

Previous categorical work [49] on cleavages and split fibrations has introduced the idea of a *lifting category*, the category of system states which restrict to a particular state of a chosen subsystem. We extend this idea to define the category of system behaviours which restrict to a particular behaviour t of a subsystem P. In keeping with extant work [49], we will refer to this category as $\mathcal{E}(t)$.

Definition 6.30. *For any behaviour t of the subsystem P identified by a view functor K, the category $\mathcal{E}(t)$ consists of*

- *objects which are those operational morphisms $r : R \rightarrow R'$ of* ***Failure Tolerance(A)*** *for which $K^*(r) = t$*
- *morphisms $m : r_1 \rightarrow r_2$ where $r_1 : R_1 \rightarrow {R_1}'$ and $r_2 : R_2 \rightarrow {R_2}'$ are objects of $\mathcal{E}(t)$, and m is a pair of morphisms $(m_1 : R_1 \rightarrow R_2, m_2 : {R_1}' \rightarrow {R_2}')$ in* ***Failure Tolerance(A)*** *such that*

$$m_2 \circ r_1 = r_2 \circ m_1$$

and $K^*(m_1) = K^*(m_2) = id$

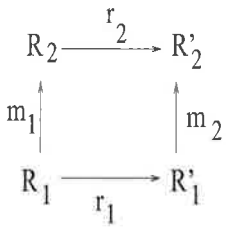

Figure 6.4: This commutativity is required for morphisms (m_1, m_2) in $\mathcal{E}(t)$

Figure 6.4 shows a morphism in the category $\mathcal{E}(t)$, where this diagram commutes. Here, m_1 and m_2 are both restoration morphisms, although Definition 6.30 allows these to be any morphisms which restrict to *id* on the subsystem P.

The $\mathcal{E}(t)$ category enables us to identify those behaviours r_1 which correspond to a particular subsystem behaviour t. In the following section, we produce some mathematical constructions upon $\mathcal{E}(t)$ which we will later apply to the security system of Section 6.3.1. These constructions will enable us to identify the properties $\mathcal{E}(t)$ must possess to minimize the effect of system failure on the behaviour t.

6.4.4 Degrees of Failure Tolerance

In this section, we present some different degrees of failure tolerance within a system. In each case, we refer to a Failure Tolerance system, where the category of states of this system is denoted **Failure Tolerance(A)**. We also identify an essential subsystem of this system, and interpret this subsystem as a Basic State system. That is, all information about this subsystem is visible under the interface represented by A. For a given behaviour t of this essential subsystem, we will use the category $\mathcal{E}(t)$ to express degrees of failure tolerance of the system with respect to this behaviour t. The different degrees of failure tolerance permit us to make different deductions about the system behaviours which must have occurred if t has been observed in the presence of system failure.

Definition 6.31. *A Failure Tolerance system demonstrates* Conditional Failure Tolerance *with respect to a behaviour $t : T \to T'$ of the essential subsystem if for any operational morphism $r : R \to R'$ where $K^*(r) = t$, and for any restoration morphism $m_1 : R \to R_2$, there exists an operational morphism $r' : R_2 \to R'_2$ which is visibly identical to r.*

Note that if r' is visibly identical to r, then r' must itself restrict to t, since the subsystem is interpreted as a Basic State system. Conditional Failure Tolerance is referred to as CFT, and is depicted in Figure 6.5. When couched in terms of the $\mathcal{E}(t)$ category, CFT is the property that, for any restoration

morphism $m_1 : R \rightarrow R_2$ of **Failure Tolerance(A)** and object $r_1 : R \rightarrow R'$ of $\mathcal{E}(t)$, there exists an object $r_2 : R_2 \rightarrow R'_2$ such that there is some morphism $m : r \rightarrow r_2$ in $\mathcal{E}(t)$.

Conditional Failure Tolerance is useful because it guarantees that system failure cannot alter the possible visible set of behaviours which restrict to t. However, CFT is often too stringent a requirement. Specifically, it requires that given *any* behaviour r restricting to t, any restoration m_1 (however inadequate) after system failure guarantees that a consistent behaviour visibly identical to r can be observed. Because this is a very strict constraint on how system failure can affect behaviours in terms of t, few systems will display Conditional Failure Tolerance. For those which do, any small alteration to the specification may cause a violation of this property, as we show in Section 6.6.

In general, we prefer to examine failure tolerance properties which are somewhat weaker. These have the advantage of holding in a greater number of systems. One method of weakening the constraints of CFT is to make use of the concept of a minimal behaviour, introduced in Section 5.3. The following failure tolerance property requires these behaviours to exist, even in the presence of system failure.

Definition 6.32. *A Failure Tolerance system demonstrates* Weakened Conditional Failure Tolerance *with respect to a behaviour $t : T \rightarrow T'$ of the essential subsystem if for any operational morphism $r : R \rightarrow R'$ where $K^*(r) = t$ and for any restoration morphism $m_1 : R \rightarrow R_2$, there exists an operational morphism $r' : R_2 \rightarrow \mathrm{R}'_2$ such that $K^*(r) = K^*(r') = t$ and*

$$r = r_2 \circ r_{min} \text{ and } r' = \mathrm{r}'_2 \circ \mathrm{r}'_{min}$$

for some visibly identical morphisms $r_{min} : R \rightarrow R_m$ and $\mathrm{r}'_{min} : R_2 \rightarrow R_n$.

Weakened CFT is also depicted in Figure 6.5. This degree of failure tolerance is less strict than CFT, as it implies that the set of system behaviours restricting to t can be affected by system failure. However, it does limit the effect that any system failure can have. To see this, first suppose the system is in a state R from which it is possible to observe a behaviour r restricting to t. Furthermore, suppose a system failure and subsequent state restoration puts the system in a visibly identical state R'. Weakened CFT states firstly that there exists a behaviour r' from R' which restricts to t and secondly that r and r' share some common visible behaviour. In contrast, CFT would have required that r and r' be completely visibly identical. Thus, we see that under Weakened CFT the ability to observe t cannot be affected by the presence of system failure, and the ways in which t can be observed must share some common minimal characteristics.

Weakened CFT can be considered as a generalisation of CFT. They both guarantee that system failure cannot affect the ability to observe the behaviour

t, but Weakened CFT places fewer restrictions on *how* t may be observed. We can define degrees of Weakened CFT, dependent on the length or complexity of the common visible behaviour r_{min} and r'_{min} shared by any r and r' as in Definition 6.32. This can be done simply by noting that a 'longer' common visible behaviour denotes a stricter degree of failure tolerance. That is, the longer this common visible behaviour, the greater the visible similarity between r and r', and hence the lesser the effect of system failure. Furthermore, we can relate this to CFT in the following way.

Suppose Weakened CFT holds in a system, and the morphisms r, r' are the morphisms of those names in Definition 6.32. These morphisms r and r' then share a common minimal visible behaviour, which is realised as the morphisms r_{min} and r'_{min} respectively. Because r_{min} and r'_{min} are visibly identical, they restrict to the same subsystem behaviour t_{min}, a consequence of the fact that the essential subsystem is interpreted as a Basic State system. Furthermore, by the definition of composition in the categories **Failure Tolerance(A)** and $\mathbf{States}_{ANC}$, the behaviour t can be written as the composition $t = t_2 \circ t_{min}$ for some morphism t_2 in $\mathbf{States}_{ANC}$. Because this holds true for every such pair r and r', we can choose the shortest of these t_{min} (which will be observed during the common visible behaviour for *every* such pair r and r') and denote it t_{trans}. CFT will then hold with respect to t_{trans}. That is, Weakened CFT implies CFT holds for a more restricted subsystem behaviour. This means that Weakened CFT for some behaviour t implies that there is a behaviour t_{min} for which $\mathcal{E}(t_{min})$ demonstrates the necessary characteristics for CFT. The composition of the category $\mathcal{E}(t_{min})$ provides us with a formal means of assessing the strictness of this particular application of Weakened CFT.

Weakened CFT holds in more systems than CFT does, precisely because it represents a weaker property. However, Weakened CFT and CFT share the same drawback that they are relatively hard to enforce in a specification. This is because both properties imply that the ability to observe t is dependent only upon the visible values in a system state. This means that neither property is suitable for analysing situations where the system behaviour depends upon the variables hidden under an interface. The following failure tolerance property addresses this deficiency.

Definition 6.33. *A Failure Tolerance system demonstrates* Minimal Behaviour Conditional Failure Tolerance *with respect to a behaviour* $t : T \rightarrow T'$ *of the essential subsystem if for any two operational morphisms* $r : R \rightarrow R'$ *and* $r' : R_2 \rightarrow R'_2$ *where* $K^*(r) = K^*(r') = t$ *and where there is some restoration morphism* $m_1 : R \rightarrow R_2$,

$$r = r_2 \circ r_{min} \text{ and } r' = \mathrm{r}'_2 \circ \mathrm{r}'_{min}$$

for some visibly identical morphisms $r_{min} : R \rightarrow R_m$ *and* $\mathrm{r}'_{min} : R_2 \rightarrow R_n$.

The property of Minimal Behaviour CFT holds in any system and for any subsystem behaviour t, since the common visible behaviour r_{min} may be the

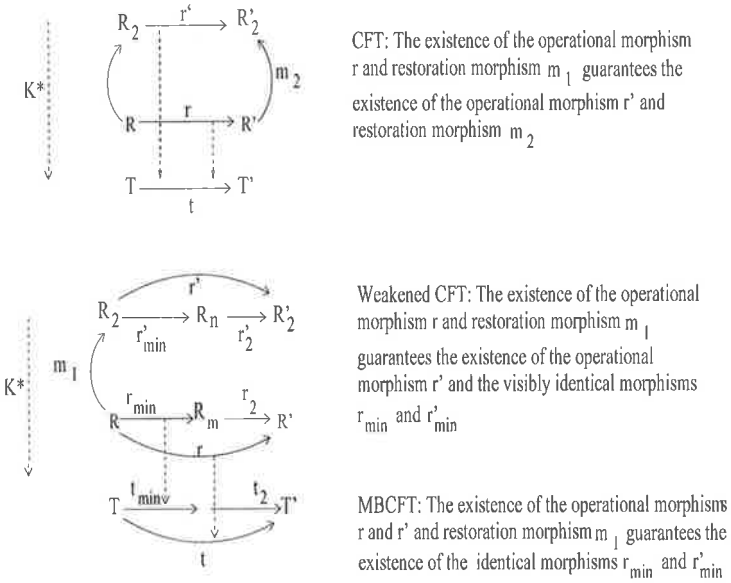

Figure 6.5: These represent CFT, Weakened CFT, and MBCFT respectively.

identity. In a system where this is the case, then we cannot deduce anything about the failure tolerance of this system.

Minimal Behaviour CFT is similar to Weakened CFT. The difference lies in the stipulation of Weakened CFT that for any behaviour r restricting to t, after a system failure and subsequent restoration of state, a behaviour r' which restricts to t and shares certain characteristics with r *must* exist. Minimal Behaviour CFT does not require that such a behaviour r' restricting to t exist after system failure, merely stipulating that *if* it exists then it must share certain characteristics with r.

Just as for Weakened CFT, we can define degrees of strictness of Minimal Behaviour CFT. These degrees are based upon the length of the minimal common behaviour r_{min} for any r restricting to t. Minimal Behaviour CFT (for some behaviour t) implies that there exists a subsystem behaviour t_{min} such that for any two objects $r_1 : R \to R_m$ and $r_2 : R_2 \to R_n$ of $\mathcal{E}(t_{min})$, there is a morphism between these objects. Specifically, this morphism of $\mathcal{E}(t_{min})$ is obtained from a pair of restoration morphisms $m_1 : R \to R_2$ and $m_2 : R_m \to R_n$. However, unlike the category $\mathcal{E}(t_{trans})$ which was generated from the property of Weakened CFT, Minimal Behaviour CFT does not allow us to conclude anything about which morphisms r will be objects within $\mathcal{E}(t_{min})$.

In the following two sections we show how these three failure tolerance properties can be used in an analysis of the security system of Section 6.3.1. Specifically, we will show how they can be used to restrict the effect of system failure upon the two fundamental behaviours of the security system — accepting user input to change the security level, and responding to an alert. When we apply these properties to the security system, the essential subsystem to which they refer will be the alert notification subsystem. The morphism $t : T \to T'$ will be a morphism of $\mathbf{States}_{ANC}$ which represents a consistent behaviour of the alert notification subsystem, while the morphisms r and r' will be morphisms within **Failure Tolerance(A)** representing behaviours of the wider security system.

6.5 Failure Tolerance and Security Settings

In this section, we examine how the security system of Section 6.3.1 can be made to demonstrate failure tolerance with respect to accepting input from a user. Specifically, we discuss the implications of requiring the system to demonstrate Conditional Failure Tolerance (Definition 6.31) with respect to the user's requests to alter the security level. The security levels themselves were described briefly in Section 6.3.1. Initially all behaviours, including requests to change the security level, are constrained by Security Policies 1, 2, 4 and 5 only. Security Policy 3 is added in the course of system refinement, a topic we discuss in Section 6.6.

In Section 6.5.1, we describe the consequences and advantages of enforcing CFT within this system with respect to accepting user input. Following this, we show that it is possible to require the security system to demonstrate CFT without this violating any Security Policies or unnecessarily constraining the freedom of the designer. This is a necessary step, since in general it is certainly not guaranteed that a system can demonstrate CFT without requiring alterations to the specification. In addition, Section 6.5.1 discusses a less obvious benefit to CFT. Specifically, we show that discussions of CFT in this context highlight a disparity between a user's expectations and the formal specifications. This is a common occurance when a user is provided only with informal descriptions of a system, such as those found in a user's manual. Ensuring CFT within the system provides a logical means of overcoming this disparity.

6.5.1 Applying CFT when changing security levels

As previously mentioned, we will let the category $\mathbf{States}_{ANC}$ represent the states and behaviours of the alert notification subsystem. Within the category $\mathbf{States}_{ANC}$, we will let the morphism t represent the behaviour of the alert notification system which occurs when the security settings are successfully changed. That is, t describes the behaviour of this subsystem corresponding to sending out a user's request and receiving a confirmation of success from all modules. We

now discuss the consequences of ensuring that the security system demonstrates CFT with respect to this behaviour t.

If CFT holds with respect to t, this means that a system failure that occurs between receiving input from a user and communicating this request to the sensors cannot cause the request to be denied if it would have originally been honoured. That is, if the system is in a state R in which it is capable of honouring a user's request (ie. there exists $r : R \to R'$ where $K^*(r) = t$), system failure and subsequent restoration of state cannot place the system in a state R_2 where it is incapable of honouring the request (ie. there must be a $r' : R_2 \to R_2{}'$ where $K^*(r') = t$). Furthermore, if CFT holds with respect to t then after system failure the system cannot be restored to a state where the only way of honouring this request is visibly different to the ways of honouring the request in the absence of this failure (ie. r and r' must be visibly identical). To illustrate why this is important, we consider the expectations of a user supplied with the descriptions of security settings in Section 6.3.1. Such a user would expect the response to his input to be predictable and reproducible. That is, he expects to be able to receive the same response from two visibly identical states in which he requests the same security settings change.

Ensuring CFT: Possible Implementation Constraints

In this section we show that it is possible for the security system of Section 6.3.1 to demonstrate CFT without causing a conflict with the Security Policies. Firstly, we will establish that enforcing CFT for user input does not restrict the ways in which a designer can implement this system. This must be ascertained for any system in which CFT is proposed. This is because CFT is a relatively strict system property and is therefore likely to cause conflict with some proposed implementations.

As established earlier, the morphism $t : T \to T'$ of $\mathbf{States}_{ANC}$ represents the behaviour of the alert notification subsystem in which a request for a security settings change is sent out to the modules and confirmation of a successful change received back. Given this, we can describe T as a state in which the alert notification subsystem can send such a message. In Section 6.3.1 we identified two possible ways in which this request could be sent: via low-priority broadcast, or separately to each module at the normal priority. The specification does not constrain either of these implementations to be incorrect. However, according to Security Policy 2, the security system will not always be able to send out broadcast requests. If the system is in a sufficiently high-alert state — although no metric for determining this has been provided at this stage of the specification — only high-priority messages will be sent. Ensuring that CFT holds with respect to t should not thus limit the freedom of the designer to implement user requests as being sent either by broadcast or as individual messages to each sensor.

The first difference between these two implementations of the communication of input lies in the fact that a broadcast message cannot be sent in *every* state R of the security system. Thus, the first way in which CFT might restrict the possible implementations is by making it possible to observe t from certain states only — namely, those in which a broadcast message *cannot* be sent. The second difference between these two implementations is that, as discussed in Section 6.3.1, a broadcast message will appear visibly different from a series of messages sent individually to sensors. Thus, the second way in which CFT might restrict the possible implementations is by requiring every system behaviour r which restricts to t to be visibly identical. In this case, only one of these implementation methods will be able to be used. We now show that CFT does not enforce either of these restrictions, and therefore does not constrain the possible implementations.

Addressing the first difference between these implementations, the only conclusion we can draw about the difference between high-alert and normal states (the latter being those in which broadcast messages can be sent) is that there must be some visible indication which of these states a system is in. This is stated in Security Policy 2. Thus, if the property of CFT is not to constrain the possible implementations, it should be possible to observe t from visibly different states in a system demonstrating CFT. To show this, we simply note that Definition 6.31 does *not* require that two states R and R_2 be visibly identical for there to exist morphisms $r : R \to R'$ and $r' : R_2 \to \mathrm{R}_2'$ which restrict to t. This means that ensuring CFT does *not* restrict the states in which user requests can be honoured to such a degree that there is only one feasible implementation of these requests.

We now address the second difference between these implementations — namely, that a broadcast message appears visibly different to a series of messages sent at normal priority. According to Definition 6.31, CFT requires visible equality of morphisms r and r' restricting to t only when these morphisms are observed from visibly identical states R and R_2. That is, CFT permits the existence of visibly different behaviours which correspond to a successful change of settings, provided the behaviours originated at visibly different states. Since the states in which a broadcast message can be sent are visibly different to those states in which it cannot, CFT allows a broadcast message and a family of individual alerts to both correspond to a successful change of security settings, despite being visibly different.

We have therefore ascertained that CFT permits both types of implementation, and does not cause conflict with the Security Policies defining each type.

Remark 6.34. *Ensuring Conditional Failure Tolerance for the security system of Section 6.3.1 when changing security settings does not constrain the means by which a system can send these requests.*

In this example there were only two suggested system implementations of the behaviour t. Because of this, it was relatively easy to show that CFT does not constrain the possible implementations. However, as the system complexity increases, it is inevitable that new implementations will be proposed. In each case, this process must be repeated for each proposed implementation method.

Ensuring CFT: User Expectations

The question of interactions between user requests, CFT and Security Policies illustrates a potential problem with the explanation of security settings provided to the user in Section 6.3.1. This explanation (suitable for a user-manual) does not provide an exhaustive listing of the circumstances in which each request might not be honoured. This is because the refusal to honour a request might be due to the hidden (recorded) information, and it is inappropriate for a user to be given access to this. As a result, a concise explanation of why a request was refused may not be available. Even if a request can be honoured in two different — though apparently identical — states, the way in which it is honoured may also depend upon the hidden data. For example, if two cameras have previously malfunctioned (shown in the recorded data) the response to a request to shut down all security may be visibly different to the response when this request is received in a state where the system is fully functional.

As a result, the system response to a user request can seem unpredictable and random. In general, customer requirements for security systems like these are that the system response to a user request is at least partially predictable by this user. To satisfy this requirement, a designer must develop a property which enforces this predictability, even in the case of system failure. Within this section, we will show that CFT provides this.

Specifically, we will require CFT to hold with respect to the behaviour t of the alert notification subsystem. Again, t here corresponds to the alert notification subsystem sending the request to change security settings and receiving confirmation. By ensuring CFT holds with respect to t, we ensure that all states in which a request *must* be refused will be visibly distinct from those states in which the request *might* be honoured. This means a user can predict the circumstances under which his request is certain to be denied. Additionally, CFT ensures that if a request to change security settings is made — and honoured — in two apparently identical states, then it is honoured in apparently identical ways. More formally, CFT implies that the existence of a behaviour r which restricts to t from any state depends only upon the visible variables in that state. That is, if R and R' appear identical, and if there exists a behaviour r from R corresponding to a successful settings change, then an identical behaviour from R' exists.

This means that if a request to change the settings is guaranteed to be denied, then this must be due to some reason visible to the user, rather than purely

due to hidden variables. Furthermore, in two apparently identical states, the request is guaranteed to be satisfied (if at all) in apparently identical fashions. This lets a user predict exactly how each family of sensors will be affected by his request to change settings. Finally, by definition this property (CFT) holds even in the presence of system failure. Thus, the user is insulated from the effects of this failure in terms of his interaction with the system.

6.5.2 Advantages of CFT for changing security settings

By ensuring that the security system of Section 6.3.1 displays CFT with respect to user input, we have provided this system with a degree of failure tolerance. While this is not complete failure tolerance of the system, CFT ensures that the response to the user's input is not badly affected by any system failure. Using CFT therefore has the two-fold advantage of allowing a designer to specify that the apparent behaviours after system failure should be similar to those before, while being lax enough to allow a certain degree of inaccuracy in the restoration.

As well as providing apparent failure tolerance to a user, CFT can be of use when two systems are communicating. Any inconsistent behaviour — due to system failure — of one could then affect the other. However, since CFT ensures that the appearance of any inconsistencies with respect to a given communication t are negligible, the security system is guaranteed to appear consistent in terms of certain communication with the external system. Finally, we have used CFT to provide a formal solution to a disparity between the user's expectations and the Security Policies. Such disparities are generally indications that futher documentation or information must be provided to an end-user. However, in the case of complex systems such as these, it may not be appropriate to provide the user with more information. Using CFT removes the ambiguity about when a user request should take precedence over a Security Policy.

6.6 Failure Tolerance and System Refinement

System refinement is the process of altering a specification as new requirements or goals are obtained. Typically, system refinement consists of adding constraints or extending a specification. Both of these activities can result in a loss of failure tolerance of the system in question.

Firstly, adding constraints reduces the possible behaviours which can be observed from any particular state. An incorrect restoration of state following system failure is then more likely to result in a visible difference between the possible behaviours from the original and restored states. This visible difference is caused by the addition of constraints relating visible information to hidden information. For example, Security Policy 3 of Section 6.3.1 might be added during a system refinement targetting the safety of this system. This Policy relates the possible behaviours of a door to the recorded information about

the codes which have been used to open it. Since this recorded information is not visible to the user, this is an example of future visible behaviours (such as whether or not the door can close) being dependent upon hidden information.

Secondly, system refinement may also involve adding more detail about each behaviour. As this level of detail increases, the likelihood that one of these "new" behaviours will be dependent upon the hidden information also increases. If this system refinement affects behaviours which occur in response to user input, then these affected behaviours may no longer be amenable to using CFT to provide a degree of predictability to the user. That is, the potential problem with user expectations which was caused by the informal description of the security settings may again need to be addressed.

We will now demonstrate some of the advantages of using Weakened CFT in place of CFT with respect to user input. This weakening of the failure tolerance criteria allows the system to evolve during refinement while still retaining a certain degree of consistency in the presence of system failure. In Section 6.6.1 we provide some examples of how a system demonstrating CFT with respect to a user's input (first discussed in Section 6.5) can violate this strict degree of failure tolerance when new constraints are added to the system. We then contrast this with a system demonstrating Weakened CFT with respect to the user's input, showing that this property continues to hold even after system refinement.

6.6.1 Violation of CFT during system refinement

As we have discussed, refining the system may involve adding constraints which limit the possible behaviours in any situation. In the security system of Section 6.3.1, this can be seen by the addition of Security Policy 3. If a door has been opened by the emergency personnel, this Security Policy forbids it to be closed until the sign-out code has been entered. This means that if a user requests a security settings change that would entail closing this door, the request will be denied. If the system demonstrates CFT with respect to user input, then according to the analysis provided in Section 6.5.1, an indication that this door cannot shut must be visible to the user.

In practical terms, this means that the door sensor would have to display information indicating that a sign-out code is required. That is, the sensor would have to make part of the recorded list of security codes visible to the user. However, the door sensors are designed to treat all recorded information as hidden, something which is likely to be integral to the physical design of each sensor. Adding Security Policy 3 might therefore mean that the door sensors must be redesigned if the system is to continue to demonstrate CFT with respect to changing security settings. Alternatively, Security Policy 3 and CFT could coexist within a system by ensuring that the only times a system will honour

a user's request to change settings is when it is impossible for any conflict to occur. In this case, this solution means that a user request to close a door would only be honoured if the door was already closed.

These two solutions respectively limit the possible physical implementations of the system by altering the design of the door modules, and trivialize the response to a user's request. Limiting the possible physical implementations is not an ideal solution, since any such limitation should have been identified prior to any system development. To include it at this later refinement stage would require the modules in question to be designed differently, causing a possibly costly delay. Equally, trivialising the response to user input in this way is certainly not the best solution either. It is therefore apparent that if Security Policy 3 is going to be added in the course of refinement, then we should consider implementing a weaker form of failure tolerance with respect to honouring a user's input.

We will implement this weaker form of failure tolerance by replacing CFT with Weakened CFT. As we have shown in Section 6.5.1, if the system demonstrates CFT then whether a security setting is changed can depend only upon the visible variables. Weakened CFT does not place the same limitations upon behavioural dependency, and therefore does not cause the same conflicts when new security policies (such as Security Policy 3) are added. The reduced level of failure tolerance offered by Weakened CFT is thus balanced by the ability to provide this failure tolerance even as the system evolves. To emphasise this, we will show how Weakened CFT — as opposed to CFT — can coexist with the newly-added Security Policy 3.

6.6.2 Weakened CFT: Refining the Security Settings

When refining the system to add Security Policy 3, we will also refine the pre-set security levels in terms of the security they provide. We then show how Weakened CFT can be used to advantage in the new, refined system. Specifically, we will interpret the application of Weakened CFT as a guarantee to provide the user with *approximately* the same results, regardless of system failure.

We quantify the concept of a "close enough" result when changing security settings by requiring the designer to define certain fundamental properties of each security level. These properties will be satisfied whenever the system is in the relevant security level. There may also be other, inessential, properties of each security level. However, the inability of a system to display these inessential properties does not constitute a reason for a user's request to enter that security level to be denied. For example, it may be a strict requirement for the **Maximum Security** setting that the doors be locked, and an inessential requirement that the motion detectors be activated. If the system cannot switch

the motion detectors on, this is not sufficient reason for it to refuse a request to enter **Maximum Security** level.

Clarification 6.35 (Refined Security Settings). *We propose a system refinement that the security settings of Section 6.3.1 be considered now as minimum levels of security, not absolute levels of security. The designer must identify defining characteristics of each security setting.*

A successful change to the security settings now means that *at least* the minimum level of security for the requested setting has been achieved. Typically, the essential or defining characteristics of each setting relate to the level of security offered by the most important sensors; cameras, motion detectors, and locking systems [16]. The activation of peripheral security systems such as metal detectors or heat-sensitive recording devices then corresponds to the desired, but not essential, characteristics of each security setting.

To reflect this refinement, we will make a corresponding change within the formal framework we use to model failure tolerance properties of the system. Previously, we required that the security system demonstrate CFT with respect to a morphism t of $\mathbf{States}_{ANC}$. This morphism t represented the behaviour of the alert notification system when sending user input to the security modules and receiving a confirmation. This was a very general interpretation of the morphism t, and has been appropriate up to this point since the system specification itself is relatively abstract. However, with system refinement, we must allow for detail to be added to the general behaviour t.

To this end, we now define a family of morphisms $t_{None}, t_{Min}, t_{Mod}, t_{Max}$ of $\mathbf{States}_{ANC}$, instead of the general morphism t. Each of these morphisms represents the behaviour of the alert notification subsystem when a request to change to a particular security setting is sent out, and a confirmation of the change is received back. For example, t_{Mod} represents the behaviour of the alert notification subsystem when a user request to enter the **Moderate Security** setting is sent, and a confirmation is received back from the sensors. We will now require that the system demonstrate Weakened CFT with respect to each of these morphisms, and discuss the implications of this.

Consequences of Weakened CFT and Security Settings

Ensuring Weakened CFT for the behaviour t_{Mod} (and the others of this family) means that if the system is in a state R where it is possible to change the security setting to **Moderate Security**, then a system failure and subsequent restoration cannot place the system in a state R' where it is impossible to successfully make this change. That is, if two states R and R_2 are visibly identical, the presence of a behaviour $r : R \to R'$ restricting to t_{Mod} implies the presence of a behaviour $r' : R_2 \to \mathrm{R}'_2$ also restricting to t_{Mod}.

In addition to this, Weakened CFT identifies a minimal common visible behaviour, denoted r_{min}, for all behaviours which restrict to t_{Mod} (or any other member of the family $t_{None}, ..., t_{Max}$). r_{min} describes the actions by which the essential characteristics of the requested **Moderate Security** setting are achieved. Every morphism r restricting to t_{Mod} must then contain a behaviour visibly identical to r_{min}, meaning that if r describes a successful security settings change, then *at least* the essential characteristics of this setting must have been achieved.

However, this does not mean that the behaviours $r : R \to R'$ and $r' : R_2 \to R_2'$ restricting to t must be visibly identical, as CFT would require. Rather, r and r' must contain the morphisms r_{min} and r'_{min} respectively (meaning they both make the essential changes), but may describe different inessential characteristics of the **Moderate Security** setting. That is, although the user may notice a difference in the behaviours corresponding to granting his settings change request before and after a system failure, this difference cannot affect any of the essential systems such as cameras or motion detectors. As a result, this failure tolerance property can coexist along with Security Policy 3 provided that the closing of all doors is not deemed an "essential" characteristic of any security level. This can easily be avoided, as the doors likely to have been opened by emergency personnel are generally outer doors which are not essential to the security of the building in question.

Weakened CFT and Designer Input

Requiring Weakened CFT instead of CFT allows the system to evolve without violating failure tolerance. It also allows the designer some input into how much failure tolerance is required. We have seen this in the discussion immediately following Definition 6.32, in which we note that the length of the minimal common behaviour r_{min} defines the degree of failure tolerance with respect to the behaviour t. For a morphism r restricting to t_{Mod}, this minimal behaviour r_{min} corresponds to the behaviour which achieves only the essential characteristics of the requested **Moderate Security** setting.

We can use these minimal behaviours, together with the increased detail obtained by using the family of morphisms $t_{None}, ...t_{Max}$ (instead of a single morphism t), to enforce differing levels of failure tolerance for different user requests. That is, a designer can now specify that some user requests should be less vulnerable to system failure than others. This is done by defining a more comprehensive minimal common behaviour r_{min} for some requests than for others. For example, a designer may decide that a request to place the system in a **Maximum Security** setting should be less vulnerable than a request to place the system in a **Minimum Security** setting. This can be reflected by identifying more essential characteristics of a **Maximum Security** setting than of a **Minimum Security** setting. In this way, the minimal common behaviour r_{min} of all morphisms r restricting to t_{Max} is more comprehensive than the minimal

behaviour for those morphisms restricting to t_{Min}. This means that any two behaviours r and r' which result in a satisfactory settings change to **Maximum Security** now have more in common than two behaviours which result in a satisfactory settings change to **Minimum Security**. That is, the effect of system failure has been lessened with respect to the **Maximum Security** setting.

This designer input is appropriate as the system undergoes refinement such as that described here. During any refinement the pre-set security levels will become easier to distinguish from each other. As a consequence, a customer will want to consider each level individually in terms of its failure tolerance and the guarantees it offers to a user. Allowing designer input means that this type of customization becomes possible.

6.7 Failure Tolerance and the Response to an Alert

In Section 6.3.1 we presented two fundamental behaviours of the security system: successfully responding to user input, and responding to an alert. So far, we have discussed how the first behaviour can be made failure tolerant. In this section we apply the same techniques to the second behaviour. The Security Policies explicitly describing the general response to an alert are Security Policies 4 and 5, although the other general Security Policies may affect an individual alert.

In general, there may be several steps which a sensor or module must take before it can confirm that a requested outcome was achieved. For example, suppose an alert is sent to the cameras requesting that they change their mode of operation from taking still images of each person entering a door(an operational mode suggested in [16]), to videotaping. In order for this outcome to be achieved successfully the camera module would need to

- receive this request
- change the status of all cameras to videotape
- acknowledge this notification and confirm the outcome

No module should ever confirm an outcome if this was not actually achieved, as stated by Security Policy 5. Instead, if the outcome cannot be achieved, the module should take no action, and send a denial back to the alert notification subsystem. Furthermore, Security Policy 4 requires that the actions taken to achieve an outcome must be 'predictable' in some way. This is a relatively ambiguous requirement at this level of abstraction, and may be interpreted in many different ways. Further system refinement would identify more comprehensively what the customer means by predictable in this context. For example, the different degrees of failure tolerance introduced in Definition 6.31, 6.32 and 6.33 all provide different degrees of this predictability, with additional differing degrees

of failure tolerance. We show firstly how ensuring CFT holds with respect to the response to an alert ensures that this reponse is predictable.

6.7.1 CFT and the Response to an Alert

In the following discussion, we let the morphism $t : T \rightarrow T'$ in $\mathbf{States}_{ANC}$ represent the behaviour of the alert notification subsystem whereby it sends an alert and receives a confirmation of the requested outcome. Security Policy 4 requires that any behaviour $r : R \rightarrow R'$ of the security system which encompasses this behaviour t (ie. for which $K^*(r) = t$) be 'predictable' in some way. A designer may initially decide to provide this property of predictability by ensuring that CFT holds with respect to t — thereby ensuring predictability which holds even in the event of system failure. Specifically, ensuring CFT holds with respect to t provides predictability because according to Definition 6.31, if two states are visibly identical then if it is possible to observe a behaviour restricting to t from one, the user can predict that it will be possible to observe a visibly identical behaviour restricting to t from the other. That is, the system states from which t may be observed are predictable, as are the possible ways in which t may be observed. By definition, CFT also ensures that this property of predictability holds even in the event of system failure.

However, as we have already seen, CFT is a very strict failure tolerance property. It implies that the possibility of observing t is dependent only upon the visible values in a state. This was an appropriate assumption when the behaviour t of $\mathbf{States}_{ANC}$ corresponded to transmitting user input, since any action taken in response to a user's request should be highly predictable to the user, who can only access visible information. However, in this discussion t simply represents a general successful response to an alert. The security system will not generally base the response to an alert solely upon the values of visible information, as this would render the recorded data useless. As a result, CFT is not an appropriate choice to ensure either predictability of the response to an alert, nor failure tolerance in the same situation. The unsuitability of CFT is only emphasised with the process of system refinement, as we show below.

6.7.2 System Refinement and the Response to an Alert

System refinement generally results in the addition of more constraints and Security Policies. As these are added, it becomes obvious that the actions taken from two different states to achieve a particular outcome will often be visibly different. For example, a request that the cameras change their mode of operation might be satisfied in different ways depending upon how much data the cameras have recorded. If a large portion of the camera module's memory is already in use, then switching to videotaping mode could cause the remaining portion of memory to be overrun. Standard practice in this case is for the cameras to upload any recorded data into a central repository to avoid memory overrun. In this case, switching the operation mode of the cameras requires

a lengthier or more complex behaviour than that required when the cameras need not perform this uploading. This example suggests that the 'predictable' behaviour of Security Policy 4 should be a certain minimal visible behaviour which must be performed in order to achieve the requested outcome.

Proposition 6.36. *The* predictability *required by Security Policy 4 will be interpreted as the existence of a certain minimal visible behaviour, dependent upon the requested outcome, which must be performed if this outcome is to be achieved.*

If we use Proposition 6.36 to define what 'predictability' means in the context of Security Policy 4, then these Security Policies 4 and 5 provide some simple guidelines for a designer adding new Security Policies during refinement. Specifically, if the proposed Security Policy describes a new outcome for some situation, then this proposed policy should also allow that if the outcome cannot be achieved then the system does nothing (in accordance with Security Policy 5). On the other hand, if the outcome can be achieved, then the proposed policy should allow that it will be achieved by performing *at least* a certain guaranteed, predictable behaviour (Security Policy 4 and Proposition 6.36).

Formulating these minimal behaviour requirements of new Security Policies can be a very complex procedure, as can ensuring that nothing changes if the outcome cannot be achieved. Furthermore, ensuring these conditions hold for each Security Policy added in the course of system refinement means that every possible interaction of the new policies with the old must be analysed. As a result, the new policies will inevitably be verbose and detailed, lacking the readability of the current Security Policies. Finally, we must ensure that both Security Policy 4 (as interpreted using Proposition 6.36) and Security Policy 5 hold even in the event of system failure. To do this, we must formally identify the behaviours or changes which must *not* occur, as well as those minimal changes which *must* occur under certain circumstances.

We first consider how to ensure that the required minimal behaviour can be observed whenever a successful outcome is achieved. Following this, we will consider the frame problem as it applies to the unsuccessful outcomes.

Minimal Behaviour CFT: Responding to an Alert

We have already shown in Section 6.6.2 how minimal behaviours might be used to ensure failure tolerance. Specifically, we used Weakened CFT to ensure the existence of some common minimal behaviour when entering a given security setting. Here we will use the same concept of a common minimal behaviour to satisfy the requirement of Security Policy 4 that the responses to an alert be predictable.

For the purposes of this discussion, the morphism $t : T \to T'$ of $\mathbf{States}_{ANC}$ will represent the behaviour of the alert notification subsystem when an alert is sent and a successful confirmation of the requested outcome received. In Section 6.7.1, we briefly discussed how requiring the system demonstrate CFT with respect to this morphism t results in a very constrained system without the potential for refinement. Another possibility for simultaneously providing failure tolerance and enforcing Proposition 6.36 is to require that the system demonstrate Weakened CFT with respect to t. This would ensure that the minimal behaviour of Proposition 6.36 is observed. Section 6.6.2 discusses some of the implications of weakening CFT in this way, with reference to the other fundamental behaviour: changing the security settings.

However, ensuring Weakened CFT holds instead of CFT is not appropriate when considering the response to an alert. This is because Weakened CFT ensures that if a successful response to an alert can be observed from some system state R, then a successful response to this alert can be observed from any visibly identical system state R_2. This was useful when t represented a successful response to a user's request, since as mentioned in Section 6.7.1, the system response to any such request should be predictable to the user. However, as we mentioned with respect to CFT, in general the ability to respond successfully to an alert is dependent upon more than the visible information. It is this same issue which made CFT unsuitable for ensuring predictability, just as it makes Weakened CFT inappropriate.

Instead, we will use the property of Minimal Behaviour CFT, introduced in Definition 6.33, to simultaneously provide failure tolerance and enforce the predictability described in Proposition 6.36. Specifically, we will require that the system demonstrate Minimal Behaviour CFT with respect to the behaviour t. Minimal Behaviour CFT implies that if a successful response to an alert is possible before system failure, then after system failure and subsequent state restoration, *if* a successful response to the alert is still possible, the two successful responses will share some common minimal visible behaviour. This means that all successful responses to an alert from visibly identical states demonstrate some common minimal behaviour, thus satisfying the 'predictability' condition of Proposition 6.36. However, unlike Weakened CFT, Minimal Behaviour CFT does not require a successful response to be possible after system failure simply because such a response was possible prior to failure. That is, Minimal Behaviour CFT allows for the possibility of a successful response to an alert being dependent upon hidden information, which may be restored incorrectly after system failure.

By ensuring Minimal Behaviour CFT of the security system with respect to a successful response to an alert, we are providing a formal degree of failure tolerance which satisfies Proposition 6.36 — which itself is intended to refine Security Policy 4. This Proposition requires a certain minimal visible behaviour

to be demonstrated in a successful response from two visibly identical states. Minimal Behaviour CFT offers this predictability, and additionally ensures that system failure cannot cause a violation of Proposition 6.36. We note that despite being a weaker form of failure tolerance than Weakened CFT, Minimal Behaviour CFT is still invaluable when it comes to ensuring that the alert notification subsystem is not badly affected by system failure. This is because, while Minimal Behaviour CFT permits an incorrect restoration to result in a different system behaviour (failing to successfully respond to an alert where it would previously have been successful), the fundamental mechanism of communicating via alerts is not compromised. That is, the functioning of the alert notification subsystem is not affected by the system failure.

6.7.3 The Frame Problem: CFT and Unsuccessful Requests

The discussion above has centred around the ways in which Security Policy 4 is applied when responding to an alert. However, Security Policy 5 is also relevant when sending an alert, since it describes what should happen if the requested outcome cannot be achieved. Specifically, Security Policy 5 requires that the system take no action — that is, another outcome should not be achieved in lieu of that requested. The exception to this is the approximation described in Section 6.6.2, where a "close enough" level of security to that requested is deemed to be a satisfactory outcome. In general, however, a security system will not permit these approximations, as they leave the system in a potentially unknown state.

To formally examine the implications of Security Policy 5, we let the morphism $t : T \to T'$ of $\mathbf{States}_{ANC}$ represent the behaviour of the alert notification subsystem when a request is sent out to one or more modules and a denial of the requested outcome is received back. The requirement of Security Policy 5 in this situation, that the modules in question take no action, is similar to the issues involved when requiring that "nothing else changes" [11] which form the basis of the frame problem. As has been explored in the literature, this is a difficult property to state precisely and to enforce in a system. However, if we refine Security Policy 5 to require that the system take no *visible* action when a requested outcome cannot be achieved, we can obtain a partial solution to the frame problem in this system, as we discuss below.

Proposition 6.37. *We propose a system refinement that Security Policy 5 should require — if a system cannot achieve a requested outcome — then it must take no* visible *action beyond communicating a denial to the alert notification subsystem.*

In order to address the frame problem, as illustrated by Proposition 6.37 refining Security Policy 5, we will require the security system to demonstrate CFT with respect to the behaviour t. As a result, this partial solution to the frame problem will hold even in the event of system failure.

Requiring the system to demonstrate CFT with respect to t ensures that if the system is in a state where it can refuse a requested outcome, then in any visibly identical state it can also refuse this requested outcome. Moreover, the actions taken in the process of refusal must be visibly identical. We need then only specify a single refusal r from each visibly distinct state, where r is a behaviour in which no visible action occurs. Once this is done, CFT will ensure that any r restricting to t will consist of taking no other action besides communicating a denial. This ensures that even in the presence of system failure, we may include constraints in the specification to model the frame problem without introducing any conflicts.

6.7.4 Failures During Behaviour

The final application of failure tolerance we examine determines how the system is affected by multiple system failures. While we can perform this analysis by examining each failure in turn and requiring that CFT or its variants hold for each associated behaviour, this is both time-consuming and error-prone. Instead, we seek a single failure tolerance metric which describes how well a system recovers from multiple failures, with respect to a particular behaviour t of the alert notification subsystem.

The following property describes this situation. Owing to the strictness of the property, we do not apply it to the security system of Section 6.3.1. This is because it requires a relatively detailed specification in order to judge the full effects of its application, and the informal Security Policies of the security system do not provide this level of detail.

Definition 6.38. *A system provides the* Minimal Behaviour Consistency Guarantee *for a behaviour $t : T \to T'$ if there is some operational morphism $r : R \to R'$ for which $K^*(r) = t$, and for any other morphism $r' : R_2 \to \mathrm{R}_2'$, where R and R_2 are visibly identical and where $K^*(r') = t$, there exists a unique operational morphism $\alpha : R' \to \mathrm{R}_2'$ such that $\alpha \circ r$ is operational and visibly identical to r'.*

In this property, r' need not be operational, and thus may represent an inconsistent behaviour. That is, r' may be a behaviour during which several system failures occurred, and at least one incorrect state restoration had a visible effect.

This property states that *any* behaviour r' resulting from system failure is visibly identical to a consistent behaviour $\alpha \circ r$ during which no failure occurs. Moreover, the visibly identical consistent behaviour results in the same Failure Tolerance state R' as the inconsistent behaviour r'. This means that there can be no further inconsistencies in the system stemming from any failure which occurred during r'. That is, not only does it describe failure tolerance of the system with respect to t, but it places a limit on the effects of any inconsistency which does arise.

The Minimal Behaviour Consistency Guarantee represents a stricter degree of failure tolerance than any of the definitions of Section 6.4.4. Because of this, it is harder to incorporate into a system and more likely to be violated during system refinement. However, in a system where multiple failures are likely, the inclusion of this guarantee is a means of describing to designers the level of failure tolerance which is required.

6.8 CCF, System Failure and Failure Tolerance

In this chapter we have discussed how system failure can cause inconsistencies within a system. System failure, such as a power cut, results in the complete or partial loss of all information within a system. If this information is not restored correctly, the system can be placed into an inconsistent state or exhibit inconsistent behaviour. Some information is more vulnerable to system failure than others. In general, this vulnerable information is that which has not been observed by any external system. This means that there is no way of guaranteeing that a state restoration is accurate upon this data. On the other hand, the effects of an inaccurate restoration can only be seen via the information which is visible to an observer. This means that incorrectly restoring the hidden information may have no negative effect.

In this chapter, we have used constructions upon the categories and functors of CCF to analyse the effects of system failure. In Section 6.2 we defined a *Failure Tolerance* system to be a Basic State system in which some information is hidden under an interface. This interface is modelled in CCF as a view functor A (a concept first introduced in Section 5.3) identifying the visible information. The information thus identified will then be able to be restored correctly in the event of system failure. We then created a category, **Failure Tolerance(A)**, in which we could formally describe equivalence of Basic states under this interface. Objects in this category represent Basic states as visible under the interface, while morphisms represent behaviours of the system. There are two types of morphism in **Failure Tolerance(A)**, and these correspond to the two types of possible behaviour which such a system can display. *Operational* morphisms correspond to consistent system behaviour, as might be observed in the absence of system failure. *Restoration* morphisms are those which correspond to the action of restoring visible system state after a system failure. A restoration morphism is a means of transitioning from any state R to another which appears visibly identical. This target state may or may not be equivalent to R under the interface. That is, this restoration may or may not be accurate, given the information visible to an observer.

In Section 6.3 we considered how this formal framework can be used in practice. To provide an application of the theory, we partially specified a security system, which we used as a case study throughout the discussion. Security Policies were used to describe the desired behaviour of this system, which consisted

of a central communications module and a number of sensors. We then formally introduced the concept of degrees of failure tolerance, which reflect how well a system recovers from system failure. Obviously, an ideal system should recover perfectly from any system failure, but in practical terms this is often hard to ensure. Instead, we identify certain subsystems which are considered essential to the correct functioning of the system, and require these subsystems to demonstrate a minimum level of consistency at all times. This echoes the identification of essential subsystems in the taxonomy of inconsistencies introduced in Section 4.7.

We then used the formal treatment of Failure Tolerance systems in CCF to define three major degrees of failure tolerance. These, given in Definitions 6.31, 6.32 and 6.33, were applied within our security system case study by identifying the central communications subsystem as "essential". That is, we used each of these failure tolerance properties to formally minimise the effect of system failure upon the messages sent within the security system. Finally, we showed how to use these formal constructions to ensure two fundamental behaviours of the security system demonstrated failure tolerance. Moreover, we showed how strict failure tolerance can be violated during system refinement, when new behaviours and constraints are added. We proposed solutions to this which consisted of weakening the degree of failure tolerance in different ways. As a result, we demonstrated how a system can undergo refinement while still demonstrating various degrees of failure tolerance.

Chapter 7

Conclusions

7.1 Summary

This thesis has proposed a category theoretic framework in which inconsistency management can be performed independently of the particular specification language used. In the sections below, we summarise the contributions in terms of the structural relationships between components in a system (Section 7.1.1), the basic structure and properties of the Categorical Consistency Framework (Section 7.1.2), and some applications of CCF (Section 7.1.3).

In Section 7.2 we discuss how the work of this thesis could be extended, both from a categorical and a system specification perspective. We propose further uses for CCF which, while stemming from inconsistency managment, have applications in other areas of system analysis. These include addressing conflicts which arise from using incompatible specification techniques, as well as a proposed method for identifying components which perform the same function in different systems. Finally, in Section 7.3 we discuss some benefits of the work proposed within this thesis.

7.1.1 Institutions and the Category of Theories

In Chapter 3 we addressed two fundamental requirements for a semantic basis of any modular specification language. These requirements refer to the conditions which determine whether two specifications are equal, as well as those conditions which determine whether a behaviour satisfies a given specification. We then showed that the category of theories can be used as a semantic basis satisfying both these requirements. This was illustrated by using the specification language Rosetta to provide examples. In the process, we discussed how using a formal semantic basis provides a means of analysing specifications which does not alter as the language evolves. This means that the analysis techniques do not have to be revised with each revision or update of the language, an advantage when it comes to reducing the cost of system specification.

We also demonstrated how to use the category of theories to formalise the relationships existing between components in any one system. This formalisation enables us to detect inconsistencies which arise when components interact. Moreover, such a formalisation also allows us to *predict* whether a given interaction could cause inconsistencies. As a result, potential problems can be detected early on in the design cycle.

7.1.2 The Categorical Consistency Framework (CCF)

In general, there are many acceptable responses to inconsistency within a system, and the choice of a response to any individual example will depend upon many factors. Detecting and resolving all the inconsistencies in a system often requires multiple frameworks and a variety of methods within these frameworks. This has traditionally made consistency checking a long and complex process.

To alleviate this problem, in Chapter 4 we presented the Categorical Consistency Framework (CCF), within which the most common inconsistencies can be detected, analysed and resolved.

CCF consists of a number of categories which together describe a dynamic system and its states. The two most important are the *system category* $\mathbf{C}$, which describes an entire dynamic system including the system constraints, and the *abstract state category* $\mathbf{C}_V$, which describes system elements as they appear in a single state of the system. A *model* of the abstract state category is a functor $R : \mathbf{C}_V \rightarrow \mathbf{Set}$, mapping system elements — such as variables — to the values they take in a particular state of the system. Since the abstract state category does not represent system constraints, a model R need not represent a consistent state. This freedom is one of the primary advantages of CCF, meaning that inconsistent states and systems may be formally represented and analysed.

The system category and abstract state category for any system are related by functors known as *state identification functors*, $\bar{R} : \mathbf{C} \rightarrow \mathbf{C}_V$. The state identification functors enforce the system constraints within a state R which factors through any $\bar{R}$. This is because system constraints are represented as equalities within the system category $\mathbf{C}$, which is the target category of any state identification functor $\bar{R}$. That is, these state identification functors ensure a certain degree of consistency of any state. Another aspect of consistency within a system is that shared information must be interpreted identically by all components. For example, a shared variable can have only one value at any point, no matter which components can access it. To enforce this type of consistency within a system we defined the *interaction consistency category* $\bar{\mathbf{C}}_V$ within CCF. Generating this category requires only details about the information each component shares with others, meaning that these inconsistencies can be detected as soon as designers make this information available. A state $R : \mathbf{C}_V \rightarrow \mathbf{Set}$ will demonstrate this type of consistency if R can factor through the interaction consistency category.

Of most interest, however, are the situations where a functor R fails to factor through the system category $\mathbf{C}$ or the interaction consistency category. When this occurs, we can use the category theoretic foundation of CCF to determine what alterations to the specification are necessary to allow R to factor through these categories. This categorical reasoning can then be extrapolated to the application in question, thereby determining the alterations which must be made to the system specification if states are to be consistent. The advantage of this approach is that detailed knowledge of the syntax or semantics of the language in which the application is specified is unnecessary. Instead, all the analysis can take place within the existing framework of category theory. The criteria for the two types of consistency (here termed component-wise consistency and interaction-wise consistency) are completely distinct. This means it is possible for a state R to demonstrate only one of these types of consistency. Because

CCF uses separate categories and functors to enforce each type of consistency, resolving an inconsistency of one type means that an inconsistency of the other type will not be introduced.

In Section 4.6 we formalised the analysis of inconsistencies by defining the category of states of a system. Objects in this category were defined to be functors $R : \mathbf{C}_V \rightarrow \mathbf{Set}$ representing consistent states, while morphisms were defined to be behaviours, or traces. We constrained all such morphisms $r = (R, [R]_i, R')$ to represent a sequence of consecutive states as determined by the state identification functors. This ensured that the system constraints in $\mathbf{C}$ relating one state to the next were applied to this sequence. Thus, if a morphism in this category does *not* represent a consistent behaviour, it can only be because these system constraints have not been satisfied. By allowing morphisms in this category to represent behaviours which are not consistent, we can compare consistent and inconsistent behaviours using categorical methods instead of application-specific analysis techniques.

Analysis of the category of states of systems allowed us to generate the taxonomy of inconsistencies of Section 4.7. Although such taxonomies of inconsistencies have been presented before, the category theoretic nature of this approach allows us to formally weigh the adverse effects of any particular inconsistency. For example, we examined inconsistent behaviours by analysing and comparing the morphisms which represented these behaviours and the morphisms which represented consistent behaviours. In all cases, we suggested categorical methods of resolving the inconsistencies, once their effect had been determined. For example, to resolve component-wise consistency we showed how to first identify the relevant constraints, and then remove those which will cause the least disruption to the system. We also discussed how using CCF enables a designer to identify certain subsystems or behaviours which are deemed essential to the correct operation of the system. Any inconsistency management can then be undertaken with the importance of these behaviours or subsystems in mind. In this way, the one general framework can be tailored for individual systems. This avoids the problems of an 'over-general' metric.

7.1.3 Applications of CCF

In Chapter 5 we demonstrated how different types of systems can be represented in CCF. By using CCF to model a range of systems, we also demonstrated the applicability of this framework to a variety of real-world problems. For example, when a designer is attempting to deduce the effect of removing or altering a component within a system, he cares more about the interconnection of the components than the actual values observed during any simulation. To model this scenario, we constructed categories within CCF which allowed us to distinguish system behaviours based upon the interconnection of components. This demonstrated how CCF can be tailored to address the precise problem in

which we are interested. Another example of how CCF can be used to model a variety of systems is shown in Section 5.4, where we constructed categories to represent systems which communicate using input/output parameters.

By using CCF to model a variety of systems, we are able to relate one type of system to another. For example, we showed how the alternative state definitions we present in Sections 5.2 and 5.4 both extend or modify the original Basic State definition. This provides a basis for comparing those systems which display characteristics of several of the different types. In addition, explicitly relating the different types of systems means that any analysis of one type of system can be extrapolated to another. For example, in Section 5.3 we used Transition History systems to show the importance of minimal underlying behaviours when considering subsystems. These are the shortest, or least disruptive, behaviours of the entire system that allow a user to observe a particular behaviour within a subsystem. We showed how these minimal behaviours can be of use when solving a Rubik's Cube (which we specify using a Transition History system), but we can also make use of this work when examining systems of other types.

Finally, in Chapter 6 we considered inconsistencies which are due to system failure. Inconsistencies of this type arise when external circumstances, such as power failure, cause information to be lost. If this information is restored incorrectly afterwards then the system might behave inconsistently. We examined this situation in the case of systems which make use of an interface. An interface hides some information, making the rest visible to an external system, such as a human observer. In the event of system failure and consequent loss of all system information, the information which was visible under the interface can be restored correctly. This is because the external observer is assumed not to be affected by the failure, and hence the observations made just prior to the failure are still valid. By contrast, the information hidden under the interface is not guaranteed to be restored correctly, since it has not been recorded anywhere but in the affected system. However, in the case of a system which uses an interface, the user is only concerned about the restoration of an *equivalent* state under the interface. In Section 6.2 we used a view functor to model an interface, and thereby to construct those categories within CCF necessary for the analysis of equivalent states. We then used the definitions and techniques introduced in Chapters 4 and 5 to produce a category of states in which equality is equivalence under the interface.

In the final sections of Chapter 6 we used this category of states to formally analyse the inconsistencies which can arise as a result of system failure. This was illustrated by a case study of a security system, a common example presented in the literature. We first identified a subsystem of this security system that we deemed essential to its functioning, then returned to the concept of minimal underlying behaviours to define degrees of consistency of this subsystem. In general, the associated disadvantage of complete failure tolerance within a

system is the difficulty in adequately specifying this property. We showed how strict degrees of failure tolerance can fail to persist in a system which is undergoing refinement, and suggested some weaker tolerance properties that are more generally applicable. By addressing this topic using the formal basis of CCF, we were able to present these degrees of failure tolerance in a coherent manner, especially in terms of the relationships they bear to each other.

7.2 Further Work

The work presented in this thesis has been intended to catalog and discuss the inconsistencies which arise at any stage throughout the specification of systems. However, there are also inconsistencies which arise at other portions of the software design lifecycle. The proposed future work lies in two broad areas related to this. First, we anticipate the generation of a similar categorical or formal framework to address those conflicts which can arise from specifying systems in different notations, or where multiple designers have different priorities for the goals of a system. Such a framework would need to be capable of translating one notation into another, while allowing for differing levels of abstraction. Secondly, the proposed framework would have to contain rules for re-prioritising system goals where these conflict. It is anticipated that quasi-classical logic will be used for this. This logic system will allow us to reason even in the presence of inconsistencies, a necessary step if we are to incorporate conflicting specifications.

Another area of interest related to inconsistency management is the consideration of issues which arise when a component is removed or replaced by another. In this case, if the two components in question demonstrate bisimilarity, we say that they are interchangeable. However, in some cases we may be satisfied with degrees of bisimilarity, just as we were satisfied with degrees of consistency in the presence of system failure. For example, two components `f1` and `f2` which differ only in that `f1` contains an unconstrained variable x not included in `f2` cannot be represented by bisimilar specifications. However, for system specification purposes, these two components are similar enough to be interchanged. We propose using CCF to generate the categories of states of each component, being denoted $\mathbf{States}_{f1}$ and $\mathbf{States}_{f2}$. We can then determine the relationship between these categories by using aspects of categorical equivalence.

In general, ascertaining whether categorical equivalence exists between $\mathbf{States}_{f1}$ and $\mathbf{States}_{f2}$ provides information about the non-determinism demonstrated by the specifications of `f1` and `f2`. The simplest way to analyse categorical equivalence for systems is to use a 'reduced' category $\mathbf{States}_{Red}$. That is, if $\mathbf{States}_{f1}$ and $\mathbf{States}_{f2}$ are equivalent categories, then we are interested in the least complex category $\mathbf{States}_{Red}$ such that $\mathbf{States}_{Red}$ is equivalent to $\mathbf{States}_{f1}$ and $\mathbf{States}_{f2}$. Here, the least complex category $\mathbf{States}_{Red}$ is likely to be generated from the category which represents only those variables which are common

to both `f1` and `f2`. The existence of $\mathbf{States}_{Red}$ then guarantees that, up to equivalence, component `f1` and `f2` act identically. That is, the degree of non-determinism offered by one component is the same as that offered by the other. Should such a category $\mathbf{States}_{Red}$ exist, we can use it to examine the behaviours of both `f1` and `f2`, and thereby ignore the additional unconstrained variable $f1.x$ in the example given above.

Establishing that two similar components demonstrate the same degree of non-determinism allows us to use them interchangeably in specifications. By phrasing all interactions in terms of the categorical equivalence functors, or even the related adjoints, we can establish which components are interchangeable. This means that the user need not define an appropriate interface in order to discover if two components are bisimilar. Further work would formalise this definition, as well as consider the notion of degrees of equivalence. This will be translated into system specification terms, thus creating a list of common refinements which may be made to a component before the altered version is deemed no longer equivalent.

7.3 Benefits of this work

The major advantage of CCF is that it provides us with a single coherent framework for formally identifying, categorising and analysing inconsistencies. We have used the specification language Rosetta to demonstrate how CCF may be used when developing a specification language, both in terms of defining the semantics, and in ensuring that these semantics offer sufficient functionality. The advantage of the resulting formal semantics is that potential inconsistencies can be detected early, thereby requiring fewer revisions of the specification. For example, without a formal framework, interaction-wise consistency can only be detected relatively late in the specification cycle. This is because all components would have to be specified to a level of detail at which it is feasible to combine them. With CCF, however, we can use the interaction-wise consistency category to identify any conflicts before the components are specified in detail. CCF also allows the representation of inconsistent states and behaviours, in contrast to many of the existing semantic bases for specification languages. The advantage of this is that both consistent and inconsistent systems may be specified and compared within this framework, meaning we can obtain a formal representation of the causes and appearances of inconsistencies. This leads to the taxonomy of inconsistencies and their potential solutions, which we proposed in Section 4.7.

Furthermore, the results obtained using CCF are compatible with those obtained using other frameworks. For example, we can use CCF and the category of theories to obtain the necessary graphs for analysing a system using **Span(Graph)**. The analysis we perform using the frameworks presented in this paper has a corresponding analysis performed using **Span(Graph)**. Similarly, techniques developed to address the database view update problem have

been used in this work in conjunction with CCF to identify minimal system behaviours. The use of existing frameworks or techniques in this way allows a user accustomed to one particular area of research to apply existing work in the new context of system specification.

CCF also allows designers to have input into the categorisation or resolution of inconsistencies by permitting the identification of "essential" behaviours, states and subsystems. An inconsistency in any of these can then be weighted within the taxonomy of inconsistencies so as to be deemed more serious than other inconsistencies which do not affect the essential functions of the system. In a similar manner, CCF also permits the definition of degrees of consistency. Often — especially in the case of system failure — it is difficult to ensure complete consistency of a system. Providing a number of degrees of consistency within CCF then allows a designer to analyse how badly a system is affected, as well as identify possible methods of reducing the inconsistency within the essential subsystems. As a result, CCF can be tailored for use in individual systems, in addition to providing generalised solutions to more abstract system specification problems.

Overall, CCF provides a coherent categorical framework for inconsistency management. We have laid the groundwork in this thesis for further enhancements, such as those suggested in Section 7.2. We anticipate that the results obtained from these enhancements, along with the analysis techniques presented here, will result in more comprehensive specification languages and cleaner, less costly specifications.

Appendix A

An Introduction to Rosetta

Appendix A: An Introduction to Rosetta

The discussion we present within this thesis was motivated by the need for a semantic basis for Rosetta. With this in mind, Rosetta syntax is used for all the examples. It is intended that the conclusions of this thesis will aid the future development of Rosetta, particularly in terms of developing more functionality for the language.

This appendix contains a brief tutorial on Rosetta. Because the language is still under development, we will only describe the basic properties, most of which are also identifiable in other specification and requirements languages, such as Clear [34], VHDL [2] and Z [94]. These properties include the ability to partition specifications, the ability to constrain variables relative to state, and the presence of a data universe [38] representing datatypes. Because we discuss only these common properties, the conclusions from this work can be applied to a general class of specification languages, of which Rosetta is a member.

The basic unit of specification in Rosetta is a *facet*, which serves as a single descriptive entity. This is similar to a VHDL entity or an OBJ3 theory [37]. A facet usually consists of a description of the behaviour of a single component, or module, within a system. However, depending on the level of abstraction, a facet can also be used to express system requirements or the physical composition of a component. A modular system can then be seen to consist of a number of components, each of which is described by one or more facets. These facets consist of declarations of variables, constants and functions. A typical facet will also include a list of constraints, or axioms, which restrict the value of these variables, constants and functions.

Facets within a system can communicate with each other in a number of ways. Two standard methods of communication are using shared data, and using message passing. These are implemented in Rosetta respectively by using global variables and input/output parameters. Section A.0.4 defines some additional methods of communication which have been proposed by the developers of Rosetta. These are known as *facet interactions* and are intended to provide a reuseable means of specifying the combination of specifications.

The following system is taken from the Rosetta Usage Guide [1] and consists of two typical Rosetta facets. The purpose of this system is to specify the behaviour of an alarmclock. We discuss the syntax in detail in Section A.0.2.

A.0.1 An AlarmClock

```
domain timeTypes;
//This defines the types we will use
```

```
  time: timeType;
  hour: hourTime;
  minute: minuteTime;
  hour = INT % 24;
  minute = INT % 60;
  time=hour, minute;
....
end timeTypes

facet clockBeh(setTime: input timeType; timeIn:input timeType;
               alarmShow: input bit; soundAlarm: output bit)
               :: state-based, timeTypes;
//inputs are all from an external user except
//alarmShow, which is from facet alarmBeh.
// soundAlarm is an output parameter to alarmBeh
  priviate soundAlarm: bit; //used to tell the alarm to
                            sound
  private clocktime: timeType; // time of this clock
  displayTime: disptimeType; // time as displayed
  begin
   setclock: IF setTime = 1
             THEN
                clockTime' = timeIn AND displayTime' = timeIn;
   tock: IF setTime = 0
         THEN
            clockTime' = increment-time(clockTime));
   showtime: IF alarmShow = 0
             THEN
                displayTime' = clockTime'
             ELSE
                displayTime' = alarmTime;
   noise: IF alarmTime = clockTime
          THEN
             soundAlarm' = 1
          ELSE
             soundAlarm' = 0;
end clockBeh

facet alarmBeh(setAlarm: input bit; timeIn: input timeType;
               alarmToggle: input bit; soundAlarm: input bit;
               alarmShow: output bit)
               :: state-based, timeTypes;
//inputs are all from an external user except
//soundAlarm, which is from facet clockBeh
//alarmShow is an output parameter to clockBeh
  private alarmShow: bit; // used to tell the clock to
```

```
                              display the alarm time
  public alarmTime: timeType; // time the alarm is set for
  private alarm: alarmType; // the noise the alarm should make
  begin
   setalarm: IF setAlarm = 1
             THEN
                alarmTime' = timeIn AND alarmShow' = 1;
             ELSE
                alarmTime' = alarmTime AND alarmShow' = 0;
   sound: IF soundAlarm = 1 AND alarmToggle = 1
          THEN
             alarm = random.alarm.noise();
end alarmBeh
```

This system is composed of three entities, `clockBeh`, `alarmBeh` and `timeTypes`, which together describe the action of an alarm clock. The facet `timeTypes` provides `clockBeh` and `alarmBeh` with datatypes, and is visible to them via inheritance. Inheritance in Rosetta is known as *facet extension*, and is signalled by the line

```
facet clockBeh(...):: timeTypes
```

This is discussed further in Section A.0.5, which includes an explanation of why `timeTypes` is referred to as a *domain*.

As the comments indicate, facet `clockBeh` accepts the inputs `setTime` and `timeIn` from a user. These indicate respectively that the user would like to set the time on the alarm clock, and what time she would like to set it to. The axiom entitled `setclock` then implements the setting of the time. We note here the use of the standard 'tick' notation, to indicate the value of a variable in the next state. The axiom entitled `tock` indicates the passing of time, corresponding to ticks of the alarm clock. The axiom `showtime` ensures that the clock displays the current time, unless the user has either entered a new alarm time. In this case, the clock will display this new alarm time, which is visible via the public (global) variable `alarmTime` from the facet `alarmBeh`. The final axiom, `noise`, uses the output parameter `soundAlarm` to trigger the facet `alarmBeh` to act if the current clock time matches the time the alarm is set for.

The second facet, `alarmBeh`, is used for constraining the behaviour of the alarm function. This facet accepts the inputs `setAlarm`, `timeIn` and `alarmToggle` from a user. These indicate respectively that the user wishes to set the alarm time, the time for which it should be set, and whether or not the alarm should be turned on. If the user does wish to set the alarm, the variable `alarmTime` records the time for which it was set, and the output parameter `showAlarm` is used to trigger the `clockBeh` facet to display this new alarm time briefly. The axiom `sound` is utilised when the time the alarm is set for matches the current

clock time, an event indicated by the value of the input parameter `soundAlarm` from the facet `clockBeh`. If this is the case, and if the `alarmToggle` input from the user is 1 (indicating that the alarm is turned on), the alarm makes a noise as determined by the `random.alarm.noise()` function. The definition of this function is not shown.

There are a number of communication methods illustrated in this system. Firstly, the two facets `clockBeh` and `alarmBeh` communicate with each other by means of input/output variables. These specifically include the variables `alarmShow` and `soundAlarm`, where an output variable from one facet is received as input to another. In addition to this, the global variable `alarmTime` enables communication between the two facets. Finally, the inheritance mechanism allows both facets to access the datatypes defined in the `timeTypes` facet.

A.0.2 Facet Syntax

As we have mentioned earlier, facets consist of a number of declarations (variables, constants, functions) along with axioms which constrain the value of these declared elements. Variables and constants can be of any declared datatype, the more common of which are listed below.

- Elemental types such as `int`,`character`, `bit`, `real` etc. [1]
  While some of these are subsets of each other, they are treated in Rosetta as unique datatypes.
- Sets and sequences of elemental types
- Constructed types (such as `tree`)
- `state`, which we discuss further in Section A.0.3

We refer the reader to [1] for a fuller description of Rosetta datatypes.

Axioms take the form $t_1 = t_2$, for terms t_1 and t_2. A term is either

- A constant, variable or function
- A variable in a particular state (such as `x@s0`, x at the initial state)
- A function applied to terms. These functions may be defined by the user, or may be pre-defined.

Functions can also include conditionals, such as 'if' and 'while' expressions. This enables us to describe properties which hold dependent upon the values of other variables. This also includes the 'tick' notation to refer to the next state. We will discuss the function of state in more detail later.

To demonstrate how these elements interact, we present the following example facet. While less interesting than the alarm clock example of Section A.0.1, the simplicity of this example means that it can be more easily used to illustrate the syntax of Rosetta, and the issues of interest.

Example 1, Rosetta Facets

```
facet f1 :: state-based
  public x: int;
  private y: int;
  begin
   T0:x@s0 = 0, x@next(s0) = 0;
   T1:x'' = x+1;
   T2:y@s0 = 0, y' = y+1;
end f1
```

```
facet f2 :: state-based
  begin
   L0: x' = x+1;
end f2
```

These example facets declare some variables, x and y, of type `int`, or integer. Additionally, these facets contain axioms constraining their variables. These axioms are denoted by $T0, T1, T2$ in the case of `f1`, and $L0$ in the case of `f2`. The standard 'tick' notation refers to the value of the variable referenced in the next state of the facet. Any facet may stutter any number of times, a concept we discuss in more detail in the following section. Because Rosetta is designed as a stuttering-insensitive language, when referencing the next state by means of the 'tick' notation, we always refer to the next state which is not obtainable by an identity state transition. The initial state of each facet is represented by the symbol $s0$, which in conjunction with the tick notation defining the next state, constrains values of the variables relative to state-changes of the facets.

The effect of the axioms $T0, T1, T2$ of `f1` is to constrain the variable x to increment at every second state transition of `f1` (modulo stutters). In addition, these axioms constrain the variable y to increment at every state change of `f1`. By means of the **public** mechanism described further in Section A.0.4, the variable x is visible to facet $f2$ as well. `f2` then constrains x by means of axiom $L0$, requiring x to increment at every state change of `f2`.

If Example 1 is to represent a consistent system, the behaviours of both `f1` and `f2` must be able to co-exist. That is, we must find a way of scheduling the state-changes which permits every axiom to be satisfied. In this simple case, the obvious solution is for facet `f1` to change state twice as often in any given period of simulation time as facet `f2` does. Chapter 5 discusses these scheduling arrangements in more detail. In the following section, we examine the structural aspects of this specification in more detail.

A.0.3 State and Rosetta

One of the distinguishing characteristics of a Rosetta specification is that each facet has its own state. The axioms in a facet then constrain its variables relative to its own state. This means that facets may change state independently of each other. A second characteristic of these specification is that any facet may stutter at any point. Stuttering is defined as a state change which is not discernable, or an identity state transition. That is, a stutter is a state transition in which the target and source states are identical [91]. These types of state change occur independently of the 'tick' notation, making any Rosetta axioms stuttering-insensitive. This means that a Rosetta system may stutter at any point, and this will not be detected; nor will it ever result in inconsistency.

A.0.4 Sharing Information Between Components

In Example 1 there are several declarations which are common to both `f1` and `f2`. Firstly, both facets reference *state*, in that all variables are constrained relative to the state of each facet. However, each facet possesses its own state, rather than sharing a common state. That is, if facet `f1` changes state, this does not automatically imply that facet `f2` has done likewise. Additionally, both facets reference elements of the `integer` abstract datatype, including the constant 0 and function +. Because the facets are communicating via shared access to a variable (x) of type `integer`, we may therefore surmise that the declaration of the integers is common to both. We do not include an axiom to the effect `f1.x = f2.x`, but this is implicit in the Rosetta syntax (such an axiom may be needed in other languages). The common declaration of the integers is available to both facets via inheritance, or *facet extension*. We discuss this in more detail below.

A.0.5 Facet Extension

Facet extension is Rosetta's means of providing inheritance. In its most basic form facet extension corresponds to the act of adding elements to a facet, thereby generating a larger or more complex facet. These added elements may be data, such as extra variables or functions, or may be axioms constraining both new and old data. Because facet extension includes all the code of the original facet in the new one, it provides a means of both re-using code and of defining a shared *data universe* [38], which we discuss below, within a system. All facet extension in Rosetta is transitive.

For any modular system, the data universe consists of declarations and datatypes which are common to multiple components in the system. The data universe therefore provides a common vocabulary which the component specifications use to communicate with each other. Of course, the exact composition of the data universe is dependent upon the individual system. However, in general the data universe will consist of the datatypes, constants and functions which are

used in more than one component. For example, in most systems the abstract datatype defining the integers (including the constant 0 and functions + and −) will form part of the data universe. In Rosetta, some other elements of the data universe in many systems are the `character` datatype, the `naturals` datatype, and the axioms of first-order logic. Facets in a system can then constrain their own variables relative to the elements of the data universe. This can be seen in axiom $T0$ of Example 1 of Section A.0.2. Here, the private variable y is constrained relative to the constant 0 and function +. This enables two facets to communicate, as a variable from one facet is now guaranteed to be interpreted appropriately (relative to the shared data universe) by another facet.

However, there is a difference between sharing a definition of a datatype, and re-using a definition. In the first case, there should be only one copy of the definition within the system, else the components cannot use it to communicate. However, if we just want to re-use code for reasons of efficiency, then the re-used code should not be shared. This corresponds to the difference, for example, between Java class variables and instance variables. For example, if we wish to provide two components `clock 1` and `clock 2` with their own alarm clocks, it makes sense to have them both use the alarmclock code from Section A.0.1. However, this does not then imply that when a user sets the time on `clock 1`'s alarmclock, that facet `clock 2` should also set its clock for this time. That is, the two alarm clocks should function independently, rather than simply being part of a shared data universe.

To emphasise the difference between code reuse and code sharing, Rosetta defines two distinct types of facet extension. The first type corresponds to the traditional object-oriented notion of inheritance. Using this, facets inherit declarations and constraints from a 'parent', resulting in a class of facets all sharing common characteristics. Each facet defines its own copy of any elements inherited in this way. The second type of facet extension corresponds to the generation of a data universe. Here, two facets extending a 'parent' view the declarations and constraints in the parent as global elements. As a result, there is only one copy of any declaration from the parent, but this one copy is accessible to all 'child' facets. Rosetta provides a means of syntactically distinguishing between the two types of facet extension. This is done by creating a separate class of facets known as *domains*.

A.0.6 Domains and Facets

As we have discussed earlier, there should be only one copy of the data universe within a system. If this were not the case, then each facet could interpret its copy of the data universe differently, and there would be no common vocabulary for communication. Since the data universe is defined by facet extension, Rosetta provides a syntactical means of identifying which declarations are to be part of this 'global' data universe. This is done by placing all such declarations within

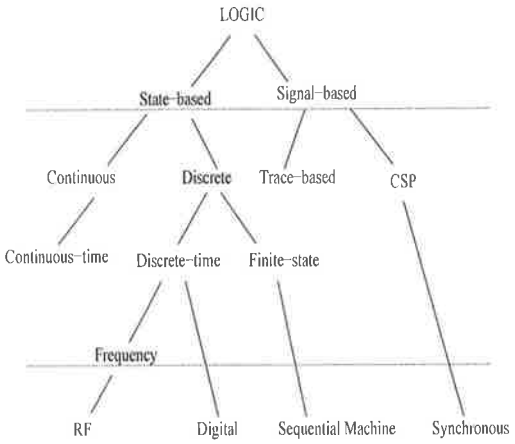

Figure A.1: The Rosetta Domain Hierarchy

special facets known as *domains*. If two facets `f1` and `f2` extend a particular domain `d1`, then they will both share one copy of any declarations in `d1`. As an example of why this is needed, suppose `f1` contains a private variable y, while `f2` contains a private variable z. If `f1` constrains `y=0` while `f2` constrains `z=0`, then knowing that the integers are part of the data universe allows us to conclude that at any given time, `y=z`. Declarations in domains can thus be seen to be similar to Java class variables.

Rosetta domains may also extend each other. This gives rise to the Rosetta Domain Hierarchy, shown in Figure A.1. At the top, the *logic* domain includes axioms of first-order logic, as well as the definition of the integers and the booleans. Nearly all Rosetta facets will extend the logic domain. This figure also shows domains extending other domains, such as the `trace-based` domain extending the `signal-based` domain. This particular extension takes place within the data universe. Facets may also extend domains, although anything defined within a facet (as opposed to a domain) is no longer considered part of the data universe. Finally, facets may extend facets, again resulting in an extension which does not provide any additional information to the data universe. However, the most common types of extension are a domain extending a domain (so adding elements to, or refining, the data universe), or a facet extending a domain.

There are some further constraints upon how facets may treat elements of the data universe. Because these elements are globally visible, any constraint placed upon them has an effect throughout the system. This can have far-reaching effects if a designer includes an erroneous axiom. For example, suppose components `f1` and `f2`, which both require a definition of the integers, are specified by

two different designers. Furthermore, suppose that this is accomplished by the integers being defined within a shared data universe. If one designer accidentally includes an axiom `0=1` in component `f1`, this will also affect the other component. Thus, the component `f2` will not work as its designer intends. Because of this, Rosetta enforces the restriction that facets may not constrain elements of the data universe. That is, a Rosetta facet can never further constrain any data originating in a Rosetta domain.

Rosetta also provides an extension mechanism by which code can simply be reused. One example of where this would be necessary is if we want to provide two components with their own copies of the alarmclock in Section A.0.1. Rosetta signals this by having the two components in question extend an alarmclock *facet*, as opposed to an alarmclock *domain*. In this way, declarations from a parent class are inherited, but each facet retains its own copy of these declarations. A facet may then constrain its own copy of a declaration without this affecting any other facet's copy. This type of facet extension can be seen most commonly in the presence of facets which extend the `state-based` facet. Within this `state-based` facet, Rosetta defines a type named `state` together with

- A constant `init` $\rightarrow$ `state`
- A function `next:  state` $\rightarrow$ `state`
- A variable `current` of type `state`

This defines an abstract datatype (a datatype and associated functions) which represents the number of state-changes (modulo stuttering) which have been undergone by the component in question. Any facet which extends this `state-based` facet will contain a copy of these definitions. This means that each of these facets contains its own `state` datatype. We note that, for historical reasons, when writing Rosetta facets we use the shorthand 'tick' notation to represent the `next` function.

A.0.7 Sharing Information Between Facets

In this appendix, we have introduced a number of ways Rosetta facets can share information. Firstly, they may do so by using global variables, as implemented using the **public** mechanism. This can be seen in Section A.0.2, where the variable x is defined in facet `f1` and visible in `f2`. Another means of sharing information is by the use of input and output variables. This was shown in the specification of an alarm clock, in Section A.0.1. In common with most specification languages, Rosetta contains a restriction that a facet may not alter the value of any of its input variables. This is discussed further in Chapter 5, wherein we provide an analysis of how consistency may be achieved amongst components which communicate using input/output parameters.

Finally, it is possible to ensure a degree of information sharing by means of *facet interactions*. These consist of predetermined methods of combining facets together to produce new facets.

A.0.8 Types of Facet Interaction

Facet interactions are generic methods for generating new facets, or describing the relationship which exists between components in a system. Rosetta provides four main types of facet interaction, although individual designers may declare their own. These basic four types are as follows.

- Facet sum
  This produces a new facet containing all the elements (data and constraints) of the two or more summands, and no additional equations or data. The most complex aspect of this is ascertaining the effects of information sharing between the summands.

- Facet implication
  Facet implication refers to the situation where one facet is said to imply another. This means that any constraint which one facet places on data is also contained within the facet which is implied. This has a converse effect on the models of these facets, which we discuss in Section 3.3.

- Facet inclusion
  Facet inclusion is one of those aspects of Rosetta still under development. Semantically, one facet may include another if the consistency of the second is reflected as a boolean value within the including facet. This then helps us define a Rosetta framework for testing consistency of Rosetta components.

- Extension
  Facet extension refers to the addition of both axioms and data elements (unlike facet implication). Facet extension is the primary means of producing new facets within Rosetta.

Appendix B

Notation

Appendix B: Notation

This list includes some of the common notation used throughout this work, along with a guide to the section where it is first introduced.

Th_{f1}: The theory representing the specification of component `f1` (Chapter 3)
CCF: The Categorical Consistency Framework, used for analysing systems. (Chapter 4)
$\mathbf{C}$: The *system category*, generated from an EA sketch and representing a system and its constraints (Section 4.3)
$\mathbf{C}_V$: The *abstract state category*, generated from an EA sketch and representing the system data visible in a single state (Section 4.4)
Basic State system: A system where states are distinguished by the current values and behaviours by the traces of the system (Section 4.2) R: A functor $R : \mathbf{C}_V \to \mathbf{Set}$, representing a state of the system (Section 4.4.3)
$\bar{\mathbf{C}}_V$: The *interaction consistency* category, representing the constraints which apply to an interaction-wise consistent state (Section 4.5.2)
D: A system model $D : \mathbf{C} \to \mathbf{Set}$, which provides a trace of the system (Section 4.3.5)
I: The quotient functor relating two categories, where one is a quotient of the other (eg. Theorem 4.25)
$\bar{R}$: The state identification functor $\bar{R} : \mathbf{C}_V \to \mathbf{C}$, used to ensure a state R factors through the system category $\mathbf{C}$ (Section 4.5.1)
States: The category of states of a Basic State system (Section 4.6)
$\mathbf{C}(Free)$: A category structurally similar to the system category $\mathbf{C}$, but representing only those system constraints which affect *every* state (Section 4.7.2)
$\mathbf{C}^k$: The *singular consistency category*, similar to the system category but representing only those system constraints which apply after k state transitions (Section 4.7.2)
$\bar{R}_{Free}$: The singular identification functor $\bar{R}_{Free} : \mathbf{C}_V \to \mathbf{C}(Free)$, used to define the number of transitions undergone by each component, and to ensure that the state R factors through the relevant singular consistency category (Section 4.7.2)
Singular States: The category of singular consistent states of a Basic State system (Section 4.7.3)
Transition History system: A system in which the number of transitions undergone can distinguish state, and behaviours are distinguished by those components which change state together (Section 5.2)
TH States: The category of states of a Transition History system (Section 5.2.6)
$\mathbf{C}_V^P$: The abstract state category of a subsystem P, usually identified by an inclusion functor $K : \mathbf{C}_V^P \to \mathbf{C}_V$ (Section 5.3.1)
$\mathbf{C}_V^I$: The *input abstraction category* for an Input State system, serving a similar purpose to the abstract state category $\mathbf{C}_V$ but not including input variables for any component (Definition 5.31)
In: The *input inclusion functor* $In : \mathbf{C}_V^I \to \mathbf{C}_V$ which identifies all system

elements which are *not* input variables of any component (Section 5.4.5)
R_{In}: A functor $R_{In} : \mathbf{C}_V^I \rightarrow \mathbf{Set}$, representing a state of an Input State system (Section 5.4.5)
Input States: The category of states of an Input State system (Section 5.4.7)
G_{fi}: The graph representing the states and state transitions of the component `fi` (Section 5.5)
$\mathbf{C}_V^{fi}$: The abstract state category of a component `fi` within the wider system (Section 5.5.2)
I_{Rep}: A functor $I_{Rep} : \mathbf{C}_V^{Rep} \rightarrow \mathbf{C}_V$, identifying those system elements which are part of a component `Rep` (Section 5.5.1)
Failure Tolerance system: A system in which an interface is defined, and equality of states becomes equivalence under this interface (Section 6.2)
$\mathbf{C}_V^A$: The *abstract state equivalence* category of a Failure Tolerance system, representing all those system elements visible under the interface A (Section 6.2.3)
A: A functor $A : \mathbf{C}_V^A \rightarrow \mathbf{C}_V$ representing the interface assumed to be part of a Failure Tolerance system
R^A: A functor $R^A : \mathbf{C}_V^A \rightarrow \mathbf{Set}$ representing a visible trace within a Failure Tolerance system, known as a *Failure Tolerance trace* (Definitions 6.9)
$R^A|$: The visible values of the state in which a Failure Tolerance trace originates (Section 6.2.3)
Failure Tolerance(A): The category of Failure Tolerance states of a system which uses an interface A (Section 6.2.5)
$\mathbf{States}_{ANC}$: The category of states of the alert notification component, given the case study of Section 6.3.1

Bibliography

[1] Alexander, P., Ashenden, P., Kong, C., Menon C. and Barton, D. "Rosetta Usage Guide". `www.sldl.org`, March 2006.

[2] Ashenden, P. "The System Designer's Guide to VHDL". Morgan-Kaufmann, 2002.

[3] Ashenden, P., Peterson, G. and Teegarden, D. "The System Designer's Guide to VHDL-AMS". Morgan Kaufmann Publishers, 2003.

[4] Bacon, J. "Concurrent Systems". Addison-Wesley, 1998.

[5] Baez, J. and Dolen, J. "Higher Dimensional Algebra and Topological Quantum Field Theory". Journal of Mathematical Physics, Vol. 36 pp. 6073-6105, 1995.

[6] Balzer, R. "Tolerating Inconsistency". Proceedings of the 13th International Conference on Software Engineering, IEEE Computer Society Press, pp. 158-165, 1991.

[7] Barker, D. "Requirements Modeling Technology, A Vision for Better, Faster and Cheaper Systems". Proceedings of the VHDL International Users Forum Fall Workshop, pp. 3-7, 2000.

[8] Barr, M. and Wells, C. "Category Theory for Computing Science". Prentice Hall, 1990.

[9] Barr, M. and Wells, C. "Toposes, Triples and Theories". `http://www.case.edu/artsci/math/wells/pub/ttt.html`, Springer-Verlag, 1983.

[10] Bolognesi, T. and Brinksma, E. "Introduction to the ISO Specification language LOTOS". The Formal Description Technique LOTOS, Elsevier Science Publishers, 1989.

[11] Borgida, A., Mylopoulos, J. and Reiter, R. "... 'And Nothing Else Changes': The Frame Problem in Procedure Specifications". Proceedings of the 15th international conference on Software Engineering, pp. 303-314, 1993.

[12] Broy, M. "Requirements Engineering for Embedded Systems". Proceedings of FemSys'97, 1997.

[13] Carroll, J. and Long, D. "Theory of Finite Automata with an Introduction to Formal Languages". Prentice Hall, 1989.

[14] Cerioli, M. "Algebraic System Specification and Development: Survey and Annotated Bibliography". Monographs of the Bremen Institute of Safe Systems, `http://citeseer.ist.psu.edu/cerioli97algebraic.html`, 1997.

[15] Chen, P. "The entity relationship model — towards a unified view of data". ACM Transactions on Database Systems, Vol. 1, pp. 9-36, 1976.

[16] Chubb Security Systems, `www.chubb.co.uk` and `www.chubb.com.au`, March 2006.

[17] Cumming, A. "A Gentle Introduction to ML". `www.dcs.napier.ac.uk/course-notes/sml/manual.html`, March 2006.

[18] Dahl, O.-J. and Owe, O. "Formal Development with ABEL". Formal Software Development Methods (VDM91), Springer LNCS, pp. 320-362, 1991.

[19] Date, C. "An Introduction to Database Systems". Addison-Wesley, 1975.

[20] Dourish, P. "Consistency Guarantees: Exploiting Application Semantics for Consistency Management in a Collaboration Toolkit". Proceedings of the ACM Conference on Computer-Supported Cooperative Work, pp. 268-277, 1996.

[21] Easterbrook, S. and Nuseibeh, B. "Using Viewpoints for Inconsistency Management". IEE Software Engineering Journal, pp. 31-43, 1995.

[22] Nuseibeh, B. and Easterbrook, S. "Requirements Engineering: A Roadmap". Proceedings of the Conference on the Future of Software Engineering, pp. 35-46, 2000.

[23] Ehrig, H., Fey, W. and Hansen, H. "ACT ONE: An Algebraic Specification Language with Two Levels of Semantics". Technical Report 83-03, Tech U. Berlin, 1983.

[24] Eilenberg, S. and MacLane, S. "General Theory of Natural Equivalences". Transactions of the American Mathematical Society, Vol 58, pp. 231-294, 1945.

[25] The Epoxi Project, `www.kestrel.edu/home/projects/dasada/`, March 2006.

[26] Fiadeiro, J. and Sernadas, A. "Structuring Theories on Consequence". Recent Trends in Data Type Specification, Springer LNCS Vol. 332 pp. 44-72, 1988.

[27] Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J. and Nuseibeh, B. "Inconsistency Handling in Multi-Perspective Specifications". Transactions on Software Engineering, Vol. 20, pp. 84-99, 1994.

[28] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L. and Goedicke, M. "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development". International Journal of Software Engineering and Knowledge Engineering, Vol. 2, pp. 31-58, 1992.

[29] Fiore, M., Moggi, M. and Sangiorgi, D. "A Fully Abstract Model for the II-Calculus". Information and Computation, Vol. 179, pp. 76-117, 2002.

[30] Gehani, N. "Ada: An Advanced Introduction". Prentice-Hall, 1983.

[31] Goguen, J. "Institutions: Abstract Model Theory for Specification and Programming". Journal of the ACM, Vol, 39, pp. 95-146, 1992.

[32] Goguen, J. and Burstall, R. "Some Fundamental Algebraic Tools for the Semantics of Computation, Part 1: Comma Categories, Colimits, Signatures and Theories". Journal of Theoretical Computer Science Vol. 31, pp. 175-209, 1984.

[33] Goguen, J. and Burstall, R. "Putting theories together to make specifications". Proceedings, Fifth International Joint Conference on Artificial Intelligence, pp 1045-1058, 1977.

[34] Goguen, J. and Burstall, R. "The Semantics of Clear: A Specification Language". Lecture Notes in Computer Science, Vol 86, pp. 292-332, 1979.

[35] Goguen, J. and Burstall, R. "Some Fundamental Algebraic Tools for the Semantics of Computation: Part 1". Theoretical Computer Science, Vol. 31, pp. 175-209, 1984.

[36] Goguen, J., Thatcher, J. and Wagner, E. "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types". Report RC-6487 IBM, 1976.

[37] Goguen, J. and Winkler, T. "Introducing OBJ3". Research Report SRI-CSL-88-9, SRI International, 1988.

[38] Goguen, J. and Malcolm, G. "A Hidden Agenda". Theoretical Computer Science Vol. 245, pp. 55-101, 2000.

[39] Gotel, O. and Finkelstein, A. "Contribution Structures", Proceedings of the Second International Symposium on Requirements Engineering". IEEE Computer Society Press, pp. 100-107, 1995.

[40] Haas, A. "The case for domain-specific Frame Axioms". F. M. Brown, 1987.

[41] Hamel, L. and Goguen, J. "Towards a Provably Correct Compiler for OBJ3". Programming Language Implementation and Logic Programming, Springer LNCS Vol. 844, pp. 132-146, 1994.

[42] Harel, D. "A Visual Formalism for Complex Systems". Science of Computer Programming, Vol. 8, pp 231-274, 1987.

[43] Harper, R., Honsell, F. and Plotkin, G. "A Framework for Defining Logics". Proceedings, Second Symposium on Logic in Computer Science, IEEE Computer Society pp 194-204, 1987.

[44] Hinchey, M. "Confessions of a Formal Methodist". Proceedings of the 7th Australian Workshop on Safety Critical Systems and Software, Conferences in Research and Practice in Information Technology, Vol. 15, pp. 17-20, 2002.

[45] Hoare, C. "Communicating Sequential Processes". Prentice Hall, 1985.

[46] van Horebeek, I. and Lewi, J. "Algebraic Specifications in Software Engineering: An Introduction". Springer-Verlag, Berlin, 1989.

[47] Hunter, A. and Nuseibeh, B. "Managing Inconsistent Specifications: Reasoning, Analysis, and Action". ACM Transactions on Software Engineering and Methodology, Vol 7, pp. 335-367, 1998.

[48] Integrated Security Systems, `www.accontrols.co.uk/index.html`.

[49] Jacobs, B. "Categorical Logic and Type Theory". Elsevier Science, 1999.

[50] Jifeng, He and Hoare, A. "Data Refinement in a Categorical Setting". Oxford University Computing Laboratory, Programming Research Group Technical Monograph, 1990.

[51] Johnson, M., Naumann, D. and Power, J. "Category Theoretic Models of Data Refinement". Electronic Notes in Computer Science, Vol. 45, 2001.

[52] Johnson, M., Rosebrugh, R. and Dampney, C. "View Updates in a Semantic Data Modelling Paradigm". Proceedings of the Twelfth Australasian Database Conference, IEEE Press, pp. 29-36, 2001.

[53] Johnson, M. and Dampney, C. "On the Value of Commutative Diagrams in Information Modelling". Proceedings of the Third International Conference on Methodology and Software Technology, Springer Workshops in Computer, pp. 45-58, 1994.

[54] Johnson, M., Rosebrugh, R. and Wood, R. "Entity-Relationship-Attribute Designs and Sketches". Theory and Application of Categories, Vol. 10, pp. 94-112, 2002.

[55] Johnson, M., Rosebrugh, R. "Update Algorithms for the Sketch Data Model". Proceedings of the Fifth International Conference on Computer Supported Cooperative Work in Design, pp. 367-376, 2001.

[56] Johnson, M., Rosebrugh, R. "View Updatability Based on the Models of a Formal Specification". Lecture Notes in Computer Science, Vol 2021, pp. 534-549, 2001.

[57] Johnson, M., Rosebrugh, R. "Universal View Updatability". `www.cs.mq.edu.au/mike/pub2000.html`, March 2006.

[58] W. Joyner, "Mathematics of the Rubik's Cube". Course Notes United States Naval Academy, `http://mathforum.org/library/view/8800.html`, March 2006

[59] Karlsson, J. and Ryan, K. "A Cost-Value Approach for Prioritizing Requirements". IEEE Software, Vol 14, pp. 67-74, 1997.

[60] Katis, P., Sabadini, N. and Walters, R. "Bicategories of Processes". Journal of Pure and Applied Algebra, Vol. 115, pp. 141-178, 1997.

[61] Katis, P., Sabadini, N. and Walters, R. "Classes of Finite State Automata for which Compositional Minimization is Linear Time". `http://citeseer.csail.mit.edu/katis01classes.html`, 2001.

[62] Katis, P., Sabadini, N. and Walters, R. "Span(Graph): A Categorical Algebra of Transition Systems". Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology, LNCS, pp. 307-321, 1997.

[63] Klar, M., Geisler, R. and Cornelius, F. "InterACT: An Interactive Theorem and Completeness Prover for Algebraic Specifications with Conditional Equations". Proceedings of the 11tgh Workshop on Specification of Abstract Data Types, LNCS pp. 274-288, Springer Verlag, 1996.

[64] Kong, C., Alexander, P. and Menon, C. "Defining a Formal Coalgebraic Semantics for the Rosetta Specification Language". JUCS Vol. 9, pp. 1322-1349, 2003.

[65] Lawvere, W. "Functorial Semantics of Algebraic Theories". PhD Thesis, Columbia University, 1963.

[66] Leite, J. and Freeman, P. "Requirements Validation Through Specification Resolution". IEEE Transactions on Software Engineering, Vol. 12, pp. 1253-1269, 1991.

[67] The LOTOS Research Group, "The Eludo Toolkit" `http://lotos.site.uottawa.ca/eludo/`, March 2006.

[68] Lowe, M. and Wolter, U. "Parametric Algebraic Specifications with Gentzen Formulas - from quasi-freeness to free functor semantics". Mathematical Structures in Computer Science, pp 69-111, 1995.

[69] McCarthy, J. and Hayes, P. "Some Philosophical Problems from the Standpoint of Artificial Intelligence". Machine Intelligence, Vol. 4, pp. 463-502, 1969.

[70] Maclane, S. "Categories for the Working Mathematician". Springer, 1997.

[71] Malcolm, G. and Goguen, J. "Proving Correctness of Refinement and Implementation". Technical Monograph PRG 114, Oxford University, `http://citeseer.ist.psu.edu/malcolm96proving.html`, 1996.

[72] The Mathworks, Matlab `www.mathworks.com/products/matlab`.

[73] Menon, C., Lakos, C. and Kong, C. "Towards a Semantic Basis for Rosetta". Proceedings of the 27th Conference on Australasian Computer Science, ACM, Vol 56, pp. 175-184, 2004.

[74] Menon, C., Johnson, M. and Lakos, C. "Inconsistency Management and View Updates". Proceedings of the Second International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures, ENTCS Vol. 141, pp. 27-51, 2005.

[75] Mosses, P. (ed) "Initiative for a Common Framework for Algebraic Specification". Bulletin of the EATCS, Vol. 59, `http://www/brics.dk/Projects/CoF`, 1996. 2283, 2002.

[76] Nuseibeh, B. "To Be and Not to Be: On Managing Inconsistency In Software Development". Proceedings of the 8th International Workshop on Software Specification and Design, IEEE CS Press, pp. 164-169, 1996.

[77] Object Management Group, "Unified Modelling Language", `http://www.uml.org/`

[78] Ostroff, J. "Temporal Logic for Real-Time Systems". Research Studios Press/Wiley, 1989.

[79] Petri, C. "Kommunikation mit Automaten", Bonn: Institut fur Instrumentelle Mathematik, 1962. "Communication with Automata" (English trans), New York: Griffiss Air Force Base, Technical Report RADC-TR-65, 1996.

[80] Robinson, W., Pawlowski, S. and Volkov, V. "Requirements Interaction Management". ACM Computing Surveys, Vol. 35, pp. 132-190, 2003.

[81] Robinson, W. and Volkov, V. "Supporting the Negotiation Life-Cycle". Communications of the ACM, Vol. 41, pp, 95-102, 1998.

[82] Robinson, W. and Pawlowski, S. "Surfacing root requirements interactions from Inquiry Cycle Requirements". Proceedings of the Third International Conference on Requirements Engineering, IEEE Computer Society Press, pp. 82-89, 1998.

[83] Robinson, W. and Fickas, S. "Supporting Multiple Perspective Requirements Engineering". Proceedings of the 1st International Conference on Requirements Engineering, IEEE CS Press, pp. 206-215, 1994.

[84] Rosebrugh, R., Sabadini, N. and Walters, R. "Minimization and Minimal Realization in Span(Graph)". Mathematical Structures in Computer Science, Vol. 14, pp. 685-714, 2004.

[85] Ross, D. "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Transactions Software Engineering SE, Vol 3, pp. 16-34, 1977.

[86] Rushby, J. "Formal Methods and Digital Systems, Validation for Airborne Systems". SRI International Computer Science Laboratory Technical Report SRI-CSL-83-7, 1993.

[87] Salibra, A. and Scollo, G. "A Soft Stairway to Institutions". Recent Trends in Data Type Specification, pp. 310-329, Springer LNCS, 1993.

[88] Schwanke, R. and Kaiser, G. "Living with Inconsistency in Large Systems". Proceedings of the International Workshop on Software Versions and Configuration Control, pp. 98-118, 1998.

[89] Spanoudakis, G. and Finkelstein, S. "Overlaps Among Requirements Specifications". Proceedings of the International Conference on Software Engineering Workshop on Living with Inconsistency, IEEE CS Press, 1997.

[90] Software Engineering Standards Committee of the IEEE Computer Society; *Chair* L. Tripp. "IEEE Recommended Practice for Software Requirements Specifications", IEEE Std pp. 830-1998, 1998.

[91] Valmari, A. "The State Explosion Problem". LNCS, Vol. 1491, pp. 429-528, 1996.

[92] Walicki, M. and Meldal, S. "A Complete Calculus for the Multialgebraic and Functional Semantics of Nondeterminism". ACM Transactions on Programming Languages and Systems, ACM Press, Vol. 17, pp. 394-421, 1995.

[93] Wood, D. and Wood, W. "Comparative Evaluations of Four Specification Methods for Real-Time Systems". Technical Report CMU/SEI-89 TR-36, Software Engineering Institute, Carnegie-Mellon University, 1989.

[94] Woodcock, J. and Davies, J. "Using Z: Specification, Refinement and Proof". Prentice Hall International Series in Computer Science, 1996.