Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 27–51
www.elsevier.com/locate/entcs

Inconsistency Management and View Updates

Catherine Menon[a,1] Michael Johnson[b,2] Charles Lakos[a,3]

[a] *School of Computer Science*
University of Adelaide
Adelaide, Australia

[b] *Information and Communication Sciences*
Macquarie University
Sydney, Australia

Abstract

Inconsistency management in component-based languages is the identification and resolution of conflicting constraints or expectations between the different components which make up a system. Here we present a category theoretical framework for detecting and classifying those inconsistencies which can arise throughout a simulation. In addition, the framework permits us to apply techniques developed for defining database view updates. With these, we can analyse the set of traces of a system with respect to a particular behaviour in a subsystem.

Keywords: specification, category theory, inconsistency management, Rosetta, traces, analysis

1 Introduction

In this paper we present a categorical framework for inconsistency management and subsystem analysis for system specification languages. We draw on existing work in the field of view updates of databases [10] to construct the framework, which enables us to treat both individual states and individual components as views of the dynamic system. The advantages of this are twofold. Firstly, we can identify and analyse inconsistencies with the aim of toleration. For example, an inconsistency may be persistent, enduring over

[1] Email: cmenon@ics.mq.edu.au
[2] Email: mike@ics.mq.edu.au
[3] Email: Charles.Lakos@cs.adelaide.edu.au

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2005.02.052

all or many states, yet confined within one subsystem only. Other inconsistencies may be systemic but transient (occurring in few states). The relative importance of these is, of course, dependent upon the system and user requirements, and a full discussion is beyond the scope of this paper, although some sample analysis is offered in Section 6. Secondly, we are able to use categorical methods to perform some state-space analysis with reference to individual subsystems or components. Specifically, we can examine all possible state transitions which restrict a given subsystem to a required pattern of observations, and find the most efficient amongst these.

Examples throughout this paper are drawn from Rosetta [1], a requirements and specification language currently under development at Kansas University. The primary unit of specification in Rosetta is a facet, which defines the characteristics and behaviour of one particular component. A facet is therefore the analogue of a VHDL entity or a Z schema in that it defines one component or viewpoint upon a system. Facets declare variables, datatypes and functions and define constraints upon these variables and functions. Rosetta has an inheritance mechanism, known as extension, which enables a facet to inherit declarations and constraints (known as *axioms*) from its parent facet. The other form of communication in Rosetta is via the use of shared variables, either as parameters passed between facets, or as globally accessible *public variables*. It is the interaction of the constraints from different facets upon a shared variable that is of interest here.

Previous semantic work based on Rosetta has consisted of a coalgebraic approach to components and their behaviours, introduced in [14], and an institutional approach to the interactions between facets, discussed in [16]. The former addresses the issues of behavioural equivalence by equating each facet with a coalgebra and analysing the behaviour that results when we examine the facets from different perspectives. The latter uses theories and institutions [7] to explore some of the issues that arise when facets are combined, in particular the questions raised by facet extension and change of notation. The use in [16] of presentations [8] and their corresponding theories allow us to express the concept of truth invariance under change of notation.

None of the previous semantic approaches have incorporated the ability to analyse inconsistent components. That is, we have only been able to study

systems where the constraints have been true throughout the entire analysis. Inconsistency can arise from a number of causes, including software evolution, user error, and under- or over- specification [5]. Once detected, the classification [18] of inconsistencies allows us to manage them in a timely manner, which is the subject of ongoing work in Rosetta. The framework we propose now provides the capability to detect and broadly classify the most common inconsistencies which arise in specification and requirements languages. One of the major causes of inconsistencies in systems specified in Rosetta is over-specification, where a system is overly constrained and cannot be implemented.

In addition to this, the work presented here discusses non-determinism, or under-specification, in Rosetta systems. This arises when a specification does not uniquely constrain the values of all its variables, which in the course of a simulation may lead to several possible next states from a given point.

Section 2 introduces the syntax and semantics of Rosetta, as well as providing examples which are used throughout the paper. In Section 3 we introduce the notion of a view update from the perspective of database design. In order to apply the resulting category theoretical ideas to Rosetta, we first need to associate a Rosetta specification with a category. Section 4 describes how we do this, as well as demonstrating how we consider subsystems as a view of an underlying system. We can also consider states as a view of an underlying *dynamic* system, but this requires some more structure, found in Section 5. This describes how we ensure that the framework implements the notion of state correctly, and provides criteria for different forms of consistency. In Section 6 we apply the theory to an example introduced in Section 2 to show how different types of inconsistency can be identified. Additionally, there are often cases where two states which are clearly distinguishable from one perspective (or facet) are seen from another as being identical, or behaviourally equivalent. In Section 7, we formalise this notion and introduce the idea of a canonical simulation path of a system with respect to a particular subsystem. Such a path, if it exists, represents the least amount of work the system needs to do in order to present a particular behaviour in that subsystem and remain consistent.

2 Rosetta

Rosetta axioms are of the form `t1 = t2` where `t1` is a term. A term is either

- A constant, variable or function
- A function applied to terms

- A term evaluated at a certain state sj

Terms which do not contain any components explicitly evaluated at a certain state are known as *base terms*. The simple system shown in Example 2.1 will be used throughout the paper in the construction of the semantic framework. Here we declare two facets (or components within a system), `f1` and `f2`, which have both inherited a definition of the `int` abstract datatype, including an increment function `+1`, from a parent facet (not shown). They have also both inherited a precise definition of state from a parent facet called `state-based`. This allows us to refer to the initial state of each facet (as $s0$), and the value of a variable x in the subsequent state (as x'). Facet `f1` declares a public variable x, which is within the scope of facet `f2` and a private variable y, which is not. Facet `f1` has three axioms, labelled $T0, T1, T2$, while facet `f2` has one axiom, labelled $L0$. The axiom $L0$ refers to the public variable x of `f1` and requires that the value of x be incremented each time facet `f2` changes state. Meanwhile, axiom $T0$ constrains the value of x to be 0 in both the initial state of `f1` and the state that results after one transition of `f1`. Axiom $T1$ requires that the value of x be incremented every *second* state-change of facet `f1`.

Example 2.1 *A simple Rosetta system*

```
facet f1 :  state-based        facet f2 :  state-based

public x::  int                L0:  x' = x+1;

private y::  int

T0:x@s0 = 0, x@next(s0) = 0;

T1:x'' = x+1;

T2:y@s0 = 0, y' = y+1;
```

When facets `f1` and `f2` interact with each other, all constraints on the variables they share must be satisfied in order for the system to remain consistent. This means that when one facet changes state, and in doing so alters the value of the shared variable x, then the other facet must also change state else the constraints upon x will conflict. In the resultant system state, then, both these facets will have changed state. This instantaneous change allows us to identify the causes behind component state-changes and is referred to in [2] as Δ-delay. Of course, in a general system it may be the case that there is actually no next state of the 'second' facet which makes the system consistent, and this situation is discussed further in Section 6.

As well as the question of scheduling state-changes in order to obtain a consistent system, we must be able to analyse the different types of inconsistency which can arise. A simple example is a specification of a light level for safety lights in a building, which must be constantly illuminated. The facet `switchmethod` accepts as input an argument `setting` giving the level of light required, which must be either 1, 2, or 3. A function (definition not shown) called `transform` uses this to decide how many lights should be illuminated, and this information is then output from the facet `switchmethod`. 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.

Example 2.2 *A switch mechanism*

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

Suppose there is also an alarm circuit in this part of the building drawing power, represented by the variable `power2`, and that we must constrain the total power used from these circuits to be constant:

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

However, the designer of the alarm circuit might also constrain his power requirements without consideration of any other circuits:

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

When all these facets are placed together, there are several potential inconsistencies due to shared variables. Section 6 shows how the framework presented in this paper might allow us to analyse these. These examples can be expressed easily in both Z and VHDL, which means the ensuing discussion is also applicable to this general class of specification languages.

2.1 State Definitions

For each facet extending `state-based`, Rosetta defines a type named `transition-number`, together with

(i) a constant `init:` $\rightarrow$ `transition-number`

(ii) a function `next:` `transition-number` $\rightarrow$ `transition-number`

(iii) a variable `current` of type `transition-number`

This defines an abstract datatype (a datatype and associated morphisms) which is equivalent to the natural numbers in the case of an infinite-state system. In the case of a finite-state system, this abstract datatype is equivalent to the integers modulo n. These elements allow us to express axioms relating values in one state to values in another, and cannot be extended or used for any other purpose in legal Rosetta. Any variable x of type `dtype` is implemented as a function $getx$: `transition-number` $\rightarrow$ `dtype`. The value of this function when given an argument $next^j(init)$ gives us the value of x after j statechanges of the facet in question. Of course, in an underconstrained system there may be several valid values for x after one statechange, and hence several possible next states from the current state. In this case, we say that this function $getx$ is not uniquely determined by the system.

Even if we can observe only a subset of variables, or properties of a state, this can be used as an abstraction mechanism [19]. Abstraction mechanisms often involve hiding variables, and mean we can perform analysis without requiring the entire state-space. For a given abstraction mechanism, two states are said to be *behaviourally equivalent* if they are identical under all observations visible under the abstraction mechanism in use.

Definition 2.3 *In Rosetta, a visible observation is an observation, or term, which returns a value of a type which is* not ***`transition-number`****.*

3 Background of View Updates

3.1 EA Sketches

EA, or *entity-attribute*, sketches are a way of representing data (entities) and relationships (attributes) between the data. As introduced in [13], an EA sketch of a database consists of a graph G, where the nodes of G are the data items (sets) in the database, and the edges of G represent the relationships between these items in the database. For example, in a graph representing a hospital database the nodes `Operations Performed` and `Doctors Practising`

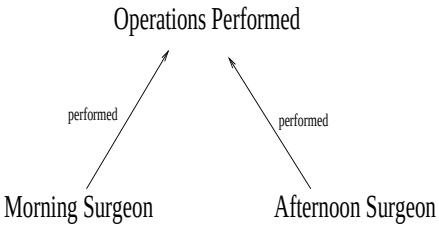

Fig. 1. The operations performed consist of the union of operations performed by surgeons in the morning and in the afternoon

may be related by means of the edge `performed by`. Each path in G then represents a query which may be made in the database, such as "What operations were performed by Doctor Smith on 1st January?". Additionally, an EA sketch contains some extra information about the relationships in the form of a family of pairs of paths $\{(p1, p2)\}$ in G where each pair $(p1, p2)$ has a common source and target. This family represents those pairs of queries in the database which produce the same result. Finally, an EA sketch identifies those relationships and data items which form those particular cones or cocones (inverted cones) known as *limit cones* and *colimit cocones*. These include maximal proper subsets and minimal containing sets of any data items and occur, for example, when one data item (set) in the database consists precisely of the disjoint union of two smaller sets. Figure 1 shows an example of a colimit cocone.

Formally, we define an EA sketch of a database as a tuple $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$. Here G is the graph associated with the database, $\mathcal{D}$ is a family $\{(p1, p2)\}$ of some of the pairs of paths in G with each pair having a common source and target, and $\mathcal{L}$ and $\mathcal{C}$ those cones (cocones) which are realised in the database as limit cones (colimit cocones). From this tuple we can generate a category C where objects and morphisms in C correspond respectively to the nodes and edges of G. Paths in G then obviously correspond to the composition of the appropriate morphisms in C. For each node n and edge e in G there is respectively a unique object and morphism in C, with the following exceptions. Firstly, the images of pairs of paths in $\mathcal{D}$ must have equal composites in C, and secondly cones (cocones) in $\mathcal{L}$ ($\mathcal{C}$) must correspond to limit cones (colimit cocones) in C. To ensure this, C is constrained to have all finite products and coproducts. This means there may now be two edges $e1, e2$ in G which correspond to the one morphism. This occurs only if the pair $(e1, e2)$ is present in $\mathcal{D}$. Thus, C encapsulates all the information contained in the tuple, in a framework which allows us to reason about the existence of functors, morphisms, and limits. Details of this are presented in [10,11,12]. A state of the database is then a limit and colimit preserving functor $D : C \to \mathbf{Set}$, the category of sets and set-valued functions. That is, given an entity (`Doctors Practising`), D

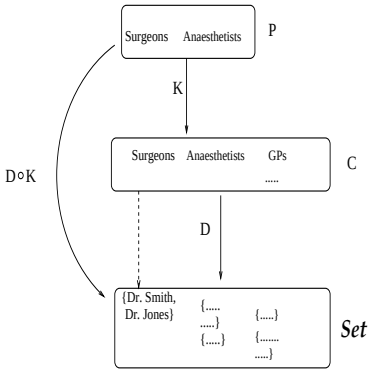

Fig. 2. A view P of an underlying system

associates this with a set (the set of all practising doctors in this database).

3.2 View Updates

In many databases, users may be given only a view of the database [4] which may not include all entities and, for those entities it does include, may not include all relationships pertaining to these. Since the relationships between entities are what govern the changes which can be made to a database, 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 if it is to remain consistent. This is exactly the same situation that arises when we examine the interaction of a subsystem with the larger system which consitutes its environment. If we define a category P consisting of those elements of C visible in the view, then we can represent this view by an inclusion functor $K : P \to C$. Composition of a model $D : C \to \mathbf{Set}$ with K then gives us a state of the view P, ie. $D \circ K : P \to \mathbf{Set}$. We denote this composition action by K^* and see that K^* then represents a functor from the category of database states D to the category of view states $D \circ K$. Figure 2 shows the interaction of views and the underlying systems. Here, the category P is the category of doctors practising in hospital, while the category C is the category of doctors registered, which may include GPs, ophthalmologists etc. in addition to anaethestists and surgeons. The mapping $D \circ K$ associates each object in P with a set. Thus we see that the object `Surgeons` in P is associated with the set containing Dr. Smith and Dr. Jones.

A view update is said to be *propagatable* [11] if there is a unique minimal change to the database which results in this update to the view [13]. We consider this minimal change, which causes the least disruption to the underlying

database, to be the *canonical update*. More details of this are available in [10] and we discuss the implications for software engineering in Section 7.

4 Rosetta Solutions

We now consider the issues specific to those views defined by Rosetta specifications. We represent a Rosetta system containing facets $f_1, ..., f_n$ by a directed graph G.

Definition 4.1 *The nodes of G, the graph corresponding to a Rosetta system, are the datatypes (such as* `integer, bit`*) declared in the facets, and the edges are the functions, variables and constants declared in each facet. Variables in a facet* `f1` *of type* `dtype` *are represented as functions with domain* `f1-transition-number` *and codomain* `dtype`*, while constants of this type are represented as functions with a null domain and codomain of type* `dtype`*. A special terminal node* `t` *serves as the null domain.*

We exclude from G the variable `current` of type `transition-number` for each facet, introduced in Section 2.1. This is because G represents an entire system statically rather than dynamically, so we have no 'current' system state. Section 4.3 describes how we achieve a dynamic perspective and where this variable is used. The state-based facet is defined such that any facet extending this declares its own `transition-number` datatype, and therefore there is a separate node `f1-transition-number` for each facet `f1` in G. However, some datatypes and functions, such as `integer`, are shared throughout the system. These make up the *data universe* [6], and ensure that all facets have a common vocabulary with which to communicate. As such, there is only one node in G to represent a datatype in the data universe. A discussion of exactly which datatypes and functions are shared within a Rosetta system is beyond the scope of this paper, but [16] provides some results from an institutional approach. We can now form an EA sketch tuple $(G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ based on this graph G.

Definition 4.2 *The paths in $\mathcal{D}$ are obtained from the Rosetta axioms, which are all of the form $t1 = t2$ for terms $t1, t2$*

This naturally gives us a pair of paths in the graph with common source and target.

Remark 4.3 *There are no axioms within $\mathcal{D}$ which explicitly constrain what state one facet is in relative to any others.*

Because $\mathcal{D}$ is generated solely from these axioms there is nothing that pertains to state synchronisation in this tuple. While such axioms can be deduced if

the system is to remain consistent, the omission of them allows us to consider inconsistent systems within this framework. In addition, such axioms would imply that some form of scheduling, or relative ordering of state changes, has already taken place, which is the optimal solution we seek using canonical state-changes.

Definition 4.4 *Elements of $\mathcal{C}$ are datatypes* `dtype` *and those morphisms $\{m_i\}$ with codomain* `dtype` *which are intended to be implemented always as an initial algebra. That is, every cocone within $\mathcal{C}$ is in fact simply a co-product of ones.*

Remark 4.5 *For any facet $f1$ the* `f1-transition-number` *datatype and its generators* `init` *and* `next` *as introducted in Section 2.1 is a co-product of ones in $\mathcal{C}$.*

For our examples, the integers are also represented by a cocone within $\mathcal{C}$. The user may, depending on the system, wish to define additional cones and cocones in $\mathcal{L}$ and $\mathcal{C}$. For example, we can use limits to ensure that a function between two sets is injective.

As introduced in Section 3, we can generate a category C from this tuple, with images of pairs of paths in $\mathcal{D}$ being commutative diagrams in C, and images of $\mathcal{L}$ and $\mathcal{C}$ being respectively limit cones and colimit cocones.

4.1 Example Sketch

For the code in Example 2.1, $\mathcal{D}$ consists of the information given in the axioms which make up the Rosetta code:

$$f1getx \circ f1next \circ f1next = +1 \circ f1getx \quad \text{(facet f1 axiom T1)}$$

$$f1gety \circ f1next = +1 \circ f1gety \quad \text{(facet f1 axiom T2)}$$

$$f2getx \circ f2next = +1 \circ f2getx \quad \text{(facet f2 axiom L0)}$$

...

These axioms do not capture the fact that facet `f2` can see the public variable x from facet `f1`. The issue of consistency (that these are not in fact separate variables and so must have the one common value in each given state) is discussed in Section 5. In Example 2.1 the only relevant cocones in $\mathcal{C}$ are the abstract datatypes `f1-transition-number` and `f2-transition-number`.

Figure 3 shows the category C for the code in Example 2.1.

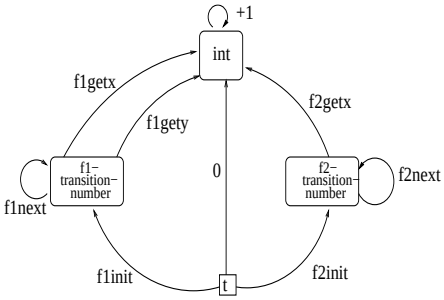

Fig. 3. The category C generated by the graph G associated with Example 2.1

4.2 Models

A model of the system is a functor $D : C \to \mathbf{Set}$ which preserves limits and colimits. D maps Rosetta datatypes to sets, and Rosetta variables, constants and functions to set-valued functions, thus providing a realisation of the categorical representation of a system. In an underdefined system there may be several valid Ds. For example, if in facet `f1` the value of an integer variable x after j transitions is undefined, then there are an infinite number of valid models D, each of which maps x after j transitions of `f1` to a different integer. Each of these Ds, therefore, provides us with a different jth state. If, however, the value of x after j transitions is constrained to be 0, then the only valid models are those D for which $D(f1getx \circ f1next^j \circ f1init) = D(0)$. A single D can then be said to provide us with one trace for each facet. Because D preserves limits and colimits, it maps a coproduct of n ones in $\mathcal{C}$ to a set consisting of n distinct elements. One consequence of this is:

Remark 4.6 *D is injective upon elements of `f1-transition-number` for each facet `f1`.*

While D provides us with a value for any variable in any state, we need further framework to analyse an individual system state.

4.3 State categories

In order to consider individual states and the transitions between them, we think of a state as simply being a view (as in Section 3) of the entire dynamic system. We do this by defining another tuple $(G_V, \mathcal{D}_V, \mathcal{L}_V, \mathcal{C}_V)$. The category generated from this, referred to as the *state category*, will represent the set of observations which can be made in any state, as well as information about how many transitions are required to obtain a state in which a given set of observations could have been made.

Definition 4.7 *Nodes of G_V are the nodes of G, while edges of G_V are the*

variables, constants and functions declared in the facets. However, unlike Definition 4.1, a variable is represented in G_V as a function with null domain, just as a constant is. We also include the variable `current` *of type* `f1-transition-number` *for each facet* `f1`*.*

$\mathcal{L}_V$ and $\mathcal{C}_V$ are identical to $\mathcal{L}$ and $\mathcal{C}$ because the constraints on cones and co-cones are not relaxed. The pairs in $\mathcal{D}_V$ are derived from $\mathcal{D}$, and the extent to which these families differ is based upon the type of analysis that we intend to do. This is because any equalities within $\mathcal{D}_V$ will apply to all states. Thus, if the user wants the ability to examine *all* inconsistent states, $\mathcal{D}_V$ will be empty. If, however, the user is not interested in analysing facets where certain inconsistencies may occur, then the axioms preventing these inconsistencies may appear in $\mathcal{D}_V$. The user will generally exclude from $\mathcal{D}_V$ any axioms which would enforce that a shared variable must have only one value at any given time, no matter how many different facets we may view it from. While such axioms can be included within $\mathcal{D}_V$, this type of error is the cause of many inconsistencies within a system and is one of the major issues of interest of Rosetta. In order to study why such an error might arise, we need a framework which permits us to examine the systems in which it does so. Section 5 discusses how we might express this situation, and consider how to correct such an error.

An example of C_V for the facet in Example 2.1 is shown in Figure 4. Here, the morphism $f1so$ `:t` $\rightarrow$ `f1-transition-number` is the equivalent of the $f1init$ morphism in the category C (Figure 3), indicating the initial state of facet `f1`. The difference in naming is to clarify the interaction of functors in Section 5. The morphism $f1current$ identifies the number of transitions facet `f1` has undergone at any given time, as introduced in Section 4. Because this is a view of a single state of the system, we do not need a function expressing the value of a variable x after every transition of a facet `f1`. (This function was present as a morphism $f1getx$: `f1-transition-number` $\rightarrow$ `int` in the category C). This is because we are only concerned about one single value of x — the value in the current state. As such, we can express this as a morphism $v1x$: `t` $\rightarrow$ `int`. Again, the change in the naming conventions is to ensure that it is obvious when we refer to a morphism in C as opposed to a morphism in C_V.

A *trace element* is then a limit- and colimit-preserving functor $R : C_V \rightarrow$ **Set**. R serves to map the elements of C_V to their values in (one of the) states

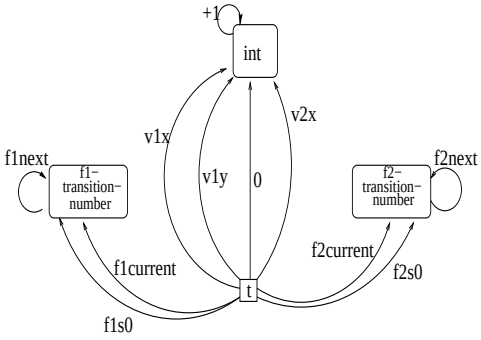

Fig. 4. The category for the abstract state structure of the system in Example 2.1

represented by this trace element. It therefore represents an assignment of values to variables together with information about how many transitions are required to get to a state in which this assignment can hold. This is achieved by the use of the variable $f1current$ in C_V (for facet $f1$) which represents how many transitions $f1$ has undergone. If for a given trace element R, we have $R(f1current) = R(f1next^j \circ f1s0)$ we say that R represents the system in a state achieved after j transitions of facet $f1$.

In a similar manner, we can isolate a subsystem P from the larger system in question and examine trace elements of this subsystem only. If P_V represents the state category for P, then a trace elements of P is of the form $T : P_V \to \mathbf{Set}$. Specifically, such a T is partly defined by a functor $K : P_V \to C_V$, as introduced in Section 3.1. Composition of K with R will then act as a view functor $T = R \circ K : P_V \to \mathbf{Set}$. It is these models that are of most interest later, when we examine the restriction of system state changes to subsystems.

5 States as Views

So far we have shown how both subsystems and states of a system can be regarded as views using this framework. However, we need to constrain the relevant views in order to identify inconsistencies and their causes. In order for a trace element R to represent a consistent state, we require firstly that R prescribe an assignment of values to variables which obeys the Rosetta constraints for a state achievable after $R(f1current)$ transitions. These constraints may take two forms:

(i) Constraints which the user wants to hold in all states, such as `x=0`, or those such as `1+2 = 2+1` which are not dependent upon the current state.

(ii) Constraints which refer to a particular state, if this is the state under consideration (such as `x@s0=0`, if we are considering state `s0`)

Secondly, in order for a state to be consistent, any shared variable cannot have multiple values at one time, regardless of how many facets can see this variable. That is, all facets must agree on the value of any variables that are visible. To enforce these ideas of consistency, we define what it means for a trace element R to be consistent.

Definition 5.1 *A trace element $R : C_V \to \textbf{\textit{Set}}$ of a system S is* consistent *if and only if the following are satisfied:*

(i) *For any axiom $(t1 = t2)$ where $t1, t2$ are Rosetta base terms (as introduced in Section 2) and $v1, v2$ are those morphisms in C_V representing these terms, we require $R(v1) = R(v2)$.*

(ii) *For any axiom $(t1 = t2)$ where $t1, t2$ are Rosetta terms which consist of terms $\hat{t1}, \hat{t2}$ evaluated after a particular number of transitions j and $v1, v2$ are the morphisms in C_V representing $\hat{t1}, \hat{t2}$, we require $R(f1current) = R(f1next^j \circ f1s0) \implies R(v1) = R(v2)$.*

(iii) *For all morphisms $v1x, v2x : t \to$ `dtype` in C_V where $v1x$ and $v2x$ respectively represent a shared variable x of type `dtype` as perceived by facets $f1, f2$, then $R(v1x) = R(v2x)$*

Definition 5.1 provides us with three different consistency conditions for an individual state. Firstly, a state R which satisfies part (i) and (ii) is a state in which each facet of the system is internally consistent. That is, in state R, any facet in the system is itself consistent in that all constraints within that facet are satisfied. However, there is no guarantee that the facets together form a consistent system in this state, as they may still be unable to agree upon the values of any shared variables. We refer to this type of consistency as *component-wise consistency*. On the other hand, a state R which satisfies part (iii) is a state in which the component facets agree on the values of all shared variables - but none of the constraints in the system beyond this are necessarily satisfied! This is referred to as *interaction-consistency*. Obviously, these refer to the extremes of possible inconsistencies. Generally the majority of constraints are satisfied and the majority of shared variables are agreed upon. It is this expectation which enables us to tolerate inconsistencies when they do arise. It is important to realise that these consistency conditions refer to an individual state only, and for a succession of states to be consistent we also require that any constraints relating the values of variables in one state to the values of variables in another be satisfied. These axioms are also present in C, in such forms as `x' = x+1`, and Section 7 explains how we ensure that an implementation of state reflects these.

As part of our treatment of a state as a view, we define an inclusion functor $\bar{R} : C_V \to C$ and then constrain $\bar{R}$ to represent a point during simulation at which each facet has undergone a certain number of state-transitions. That is, the morphism to which a given $\bar{R}$ maps $f1current$ determines the state of facet $f1$. All $\bar{R}$ must act as the identity on all Rosetta datatypes in C_V (note that these datatypes are also present in C).

Definition 5.2 *A functor $\bar{R} : C_V \to C$ is* valid *only if $\bar{R}$ is the identity on all morphisms corresponding to the common edges of G_V and G. Also, for each variable x seen by a facet $f1$, the action of $\bar{R}$ must be such that*

- $\bar{R}(f1current) = f1next^k \circ f1init \implies \bar{R}(v1x) = f1getx \circ f1next^k \circ f1init$ *where $v1x$ is the morphism in C_V representing x as seen by $f1$*

This ensures that Rosetta axioms in C originating from a facet $f1$ which constrain a variable x do in fact affect the morphism in C_V which represents x as seen by $f1$, and not a C_V morphism representing some other quantity y. It also ensures that $\bar{R}$ incorporates the correct mappings for the transition numbers so that these do correctly implement state. That is, if the $f1current$ variable indicates that k transitions have been undergone by facet `f1`, then $\bar{R}(v1x)$ must correspond to the morphism in C indicating the value of x as seen by `f1` after k transitions precisely.

$\bar{R}$ for Example 2.1 can be seen in Figure 5. Here we can see that $f1s0$, the morphism in C_V corresponding to the initial state of facet `f1`, is mapped to $f1init$, the morphism in C corresponding to the initial state of facet `f1`. This ensures that the only difference between different $\bar{R}$ is what morphism $\bar{R}(f1current)$ corresponds to, for a facet `f1`. Obviously, for some j, we will have $\bar{R}(f1current) = f1next^j \circ f1init$. If this is the case, then by Definition 5.2, $\bar{R}$ will map $v1x$ to the morphism $f1getx \circ f1next^j \circ f1init$ in C. We can then express a valid R as the composition of a system model D and such a $\bar{R}$.

Theorem 5.3 *If $R = D \circ \bar{R}$ where $\bar{R}$ is valid as in Definition 5.2, then R is a component-wise consistent state.*

Proof. Part(i) For any equality $(t1 = t2)$ where $t1$ and $t2$ are base terms (Section 2) which are Rosetta functions, constants, or function applications, there is a commutative diagram $m1 = m2$ in C, where $m1, m2$ are the morphisms in C representing the terms $t1, t2$. Letting $v1, v2$ be the morphisms in C_V representing the Rosetta terms $t1, t2$, then since by Definition 5.2 we

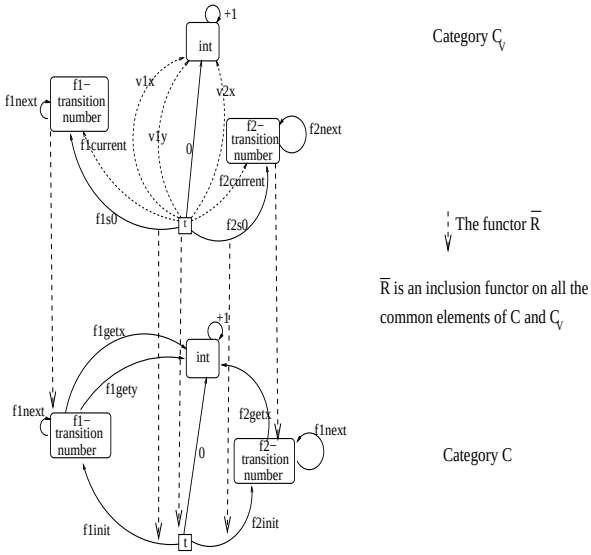

Fig. 5. The operation of $\bar{R}$ on parts of C_V

know $\bar{R}$ is the identity inclusion functor on constants and functions, we have $\bar{R}(v1) = m1 = m2 = \bar{R}(v2)$ and hence $R(v1) = R(v2)$.
If $t1, t2$ are base terms involving variables seen by $f1$, then this equality also holds because for some j, $\bar{R}(v1) = m1 \circ f1next^j \circ f1init$ and $\bar{R}(v2) = m2 \circ f1next^j \circ f1init$ by Definition 5.2, and hence $\bar{R}(v1) = \bar{R}(v2)$. For R defined as above, this then implies $R(v1) = R(v2)$
Part(ii) If there is an axiom $(t1 = t2)$ which holds in the jth state only, then without loss of generality we can say $t1$ and $t2$ consist of Rosetta base terms $\hat{t1}, \hat{t2}$ evaluated after j transitions. This means that for $m1, m2$ those morphisms in C which represent $\hat{t1}, \hat{t2}$, the equality $m1 \circ f1next^j \circ f1init = m2 \circ f1next^j \circ f2init$ holds within C. If $v1, v2$ represent the terms $\hat{t1}, \hat{t2}$ in C_V then by Definition 5.2,
$\bar{R}(f1current) = f1next^j \circ f1init \implies \bar{R}(v1) = m1 \circ f1next^j \circ f1init$ and $\bar{R}(v2) = m2 \circ f1next^j \circ f1init$. That is, $R(v1) = R(v2)$ when $R(f1current) = R(f1next^j) \circ R(f1s0)$, since we remember from Remark 4.6 that D is injective upon elements of `fi-transition-number`. □

To ensure interaction-consistency for a state R, we need to add axioms which constrain shared variables to be equal. Such axioms do not form part of the specification, since this would reduce the capability for independent construction of facets. As such, we define a new category $\bar{C}_V$ which represents the abstract structure of a state (as does C_V) but in which axioms enforcing the equality of shared variables are included. It is obvious that any model $D' : \bar{C}_V \to \mathbf{Set}$ of this category $\bar{C}_V$ will then obey these constraints.

Definition 5.4 *We define a tuple* $(G_V, \mathcal{D}_V \bigcup \mathcal{D'}_V, \mathcal{L}_V, \mathcal{C}_V)$*, where* $\mathcal{D'}_V$ *consists of all pairs* $(e1, e2)$ *where* $e1$ *and* $e2$ *represent a shared variable as observed by two different facets.* G_V*,* $\mathcal{D}_V$*,* $\mathcal{L}_V$ *and* $\mathcal{C}_V$ *are those elements of the tuple associated with* C_V*, introduced in Secion 4.3. We refer to the category associated with this new tuple as* $\bar{C}_V$*.*

That is, for a variable x shared between facets `f1` and `f2`, we include the pair $(f1.x, f2.x)$ in $\mathcal{D'}_V$ to enforce $f1x = f2x$. The category $\bar{C}_V$ is structurally identical to the category C_V, save for the addition of these equalities. We formalise this similarity below.

Definition 5.5 *Define a functor* $X : C_V \to \bar{C}_V$ *where* X *is the quotient map, or the identity modulo the addition of the axioms from* $\mathcal{D'}_V$*. We also define a functor* $Y : \bar{C}_V \to Set$*, placing no restrictions on* Y*.*

That is, for morphisms $v1x$ and $v2x$ in C_V representing a shared variable x as seen by facets $f1$ and $f2$ respectively, the axiom $X(v1x) = X(v2x)$ applies to $\bar{C}_V$. The functor Y serves as a model of $\bar{C}_V$.

Theorem 5.6 *If* $R = Y \circ X$ *where* X *satisfies Definition 5.5, then* R *is an interaction-consistent state.*

Proof. Let $v1x, v2x : t \to$ `dtype` be morphisms in C_V representing the value of a shared variable x as seen by facets $f1$ and $f2$. Then $X(v1x) = X(v2x)$ by definition and therefore $Y \circ X(v1x) = Y \circ X(v2x)$, ie. $R(v1x) = R(v2x)$.

Lemma 5.7 R *is* consistent *and* valid *iff*

(i) $R = D \circ \bar{R}$ *for some valid* $\bar{R}$ *(as in Definition 5.2) and system model* D

(ii) $R = Y \circ X$ *for* X *which satisfies Definition 5.5*

(iii) R *preserves limits and colimits*

The first two conditions of this lemma simply state that to be consistent a state R must be both component-wise consistent (Theorem 5.3) and interaction-consistent (Theorem 5.6). The final condition refers to the situation where a user has specified conflicting constraints which can be satisfied only by an implementation of some datatype as something other than an initial algebra. For example, the effect of implementing a pair of axioms such as `x=1; x=0` into C is to incorporate the axiom `1 = 0`. While this is legal in the construction of C, it is usually not what the user intended. Thus, for the strictest form of consistency of one state, we use the fact that the relevant datatypes in C_V do not have these additional equalities. Any R in this situation which is factored through a valid $\bar{R}$ can therefore not be colimit-preserving. That is, R is a

valid model of a state of the specified system when the diagram in Figure 6 commutes, and R preserves limits and colimits. By specifiying that a model T

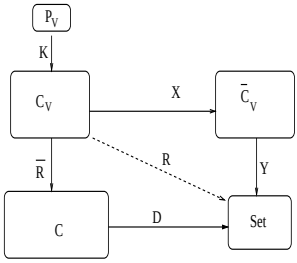

Fig. 6. Some commutativity is required for R to be valid

of a partial system P be a functor $R \circ K$, there is an obvious parallel between database views and partial systems.

6 Example Analysis

We are now in a position to examine the inconsistencies which may arise in Example 2.2. Firstly, examining facet `switchmethod` in isolation, we see that the simultaneous presence of axioms `switchmethod.T0` and `switchmethod.T2` imply that there is no state R of this system which satisfies part (iii) of Lemma 5.7. This is because in any state R the integers form an infinite coproduct of ones in C_V, yet R does not preserve this coproduct, since it is not present as an infinite coproduct within C. These persistent inconsistency errors are difficult to work around or tolerate, as it is generally unclear what the user has intended. However, they can be detected relatively easily by testing components in isolation. In this case, the removal of axiom `switchmethod.T2` is sufficient to resolve the inconsistency. On the other hand, when we place all facets together, by some means such as

```
facet security = switchmethod AND powerreq AND alarmreq;
```

we may see an additional inconsistency occur. This is due to a potential conflict between the constraints set upon the `power2` variable, depending upon what input is received from the user. Many of the otherwise valid system models D which exist fail to form part of a commuting diagram as in Figure 6, due to conflicting constraints from the alarm circuitry and light switch. In this case, a D which worked perfectly when testing in isolation (for example, one which implemented $alarmon = 20$) fails when we attempt to resolve the areas of overlap, since there exists no functor Y which may implement this. This is a different kind of inconsistency, as the issue here is that we cannot guarantee that the diagram corresponding to a particular implementation will

commute. Resolution for these is more flexible, as it is possible either to add information to the system (for example, adding axioms constraining the possible value of `alarmon`) or to relax constraints (for example, removing the axiom $A0$). In practice, such errors often indicate underspecified systems and as such the recommended practice is to further constrain the system. It is worth noting that this is possible because the nature of the problem is such that any functors that form a commuting diagram are also compatible with the other constraints.

7 Canonical Morphisms and State Progression

In order to examine traces, we now develop a mathematical representation of state transitions as progressions between valid trace elements. This enables us to compare those traces which produce some particular observations in a subsystem, or those underlying system transitions which restrict upon a particular subsystem to produce a desired behaviour. This is especially useful when discussing subsystems exhibiting critical behaviour, as we can abstract away from implementation issues to compare the family of traces which display the required observations. In particular, we are interested in the shortest possible trace, or 'most efficient' sequence of state-changes of the system which guarantee the subsystem undergoes the particular statechanges in which we are interested. This is the specification analog to the database issue of identifying propagatable views. In order to express this mathematically, we need the following categorical concepts.

7.1 Cocartesian morphisms and fibrations

Let R, R' and R'' be objects in a category L, let A be an object in a category L' and let K^* be a functor $L \to L'$. A morphism $r : R \to R'$ in L is a *pre-cocartesian* lifting [9] of $t : K^*(R) \to A$ if

(i) $K^*(r) = t$

(ii) For all other $r' : R \to R''$ in L where $K^*(r') = K^*(r)$, there is a unique $r'' : R' \to R''$ such that $r'' \circ r = r'$ and $K^*(r'') = id$.

This can be seen in Fig. 7. Note that two pre-cocartesian liftings of t will be isomorphic, and that it is also implied that $A = K^*(R')$. A cocartesian lifting is a slightly stronger property, used often in topology. To use this mathematical property in a database setting, we let objects (eg. R, R') of L be states of the underlying database, and objects of L' be states of the 'view' database. We can then say that the morphism (or state-change) r in L is the canonical update corresponding to the view update t. That is, r is the

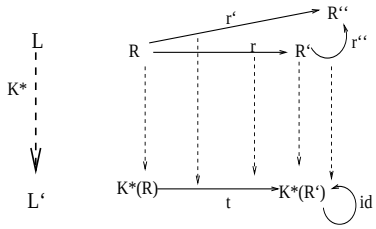

Fig. 7. A pre-cocartesian lifting

minimal underlying database changed required for us to observe the change t in the view.

7.2 Application of View Updates to Specification Languages

Each component, or family of components P, also has a particular view of the entire system and there may be many underlying system state-changes which restrict to a particular state-change of P. The immediate example is where two families of components in a system share no common data. In this case, any system state-change involving only facets in the first family will always restrict to the identity (or no observable state-change) of those facets in the second family.

Definition 7.1 *The* canonical underlying system state change*, with reference to a given state change t of a family of components P, is the underlying system state-change which restricts to t when examining P, and which posesses the 'pre-cocartesian' property introduced in Section 7.1, for the category of states of the system.*

To illustrate, in a system which models a Rubik's cube [17], we may want to know the shortest sequence of moves of the cube which allow us to observe a sequence of patterns on one face. Here, the existence of a shortest sequence is clearly dependent upon the patterns we want to see. In the case where a canonical morphism exists for every possible simulation path of a family P of components, we know exactly how much work, in terms of underlying statechanges, has to be done by the system to allow this family P to function.

7.3 Applications

For each system we define a category L where objects in L are the valid trace elements R, and morphisms in L are a way of moving between the states these trace elements represent. This then lets us implement other abstraction mechanisms by clustering distinct trace elements which are indistinguishable

under the abstraction mechanism in question to represent the same state. The example we present below does not use any abstraction mechanism. That is, two objects R, R' in L are equal iff $R = R'$ as functors. Since any valid consistent state R preserves limits and colimits, it follows that it is injective upon the infinite coproduct of ones representing the `f1-transition-number` abstract datatype within $\mathcal{C}_\mathcal{V}$ for any facet `f1`.

Lemma 7.2 *Given any valid trace element $R = D \circ \bar{R}$, there is a unique $\bar{R}$ for which this equality holds.*

The unique $\bar{R}$ can be found easily since $R(f1current) = R(f1next^j \circ f1init)$ for some unique j, and Definition 5.2 then allows us to precisely identify the action of $\bar{R}$.

There are many different definitions of morphisms, the simplest (and most restrictive) being one we use to analyse what causes facet state-changes. For example, a state-change could be driven by receiving input, or in order to retain consistency after another facet changes state and hence changes the values of some variables. The following definition emphasises those facets which change state together and the reason for this, rather than the actual states involved throughout a sequence of changes. To model this we define a morphism $r : R \to R'$ to be a tuple $(R, [\bar{R}_i], R')$, where $[\bar{R}_i]$ is an ordered list of functors $\bar{R}_i : C_V \to C$ as in Section 5. This list tells us how a system may move from some state represented by R to some state represented by R', by providing us with information about which components change state together (ie. separated only by a Δ-delay). We can then examine how certain state transitions of a subsystem affect other components, and from this determine an objective measure of the interaction between the subsystems.

In order to ensure that the progression does in fact reflect state-changes with regard to the implementation of `transition-number` within C, we place some restrictions on elements of the family $[\bar{R}_i]$. Specifically, each facet may change state at most once in each system state change, and any state-change of a facet increments the value of the `current` variable pertaining to that facet. That is, for any facet $f1$, we require either

(i) $D \circ \bar{R}_{i+1}(f1current) = (D \circ \bar{R}_i(f1current)) + 1$ or

(ii) $D \circ \bar{R}_{i+1}(f1current) = (D \circ \bar{R}_i(f1current))$

One possible definition of composition is simple concatenation of the ordered lists $[R_i]$. Thus, for $r = (R, [\bar{R}_i], R')$, and $r_2 = (R', [\bar{R'}_j], R_2)$ then $r_2 \circ r = (R, [[\bar{R}_i], \bar{R'}, [\bar{R'}_j]], R_2)$.

As introduced in Section 3, we may wish to examine those morphisms which produce a particular behaviour $t : T \to T'$ in a subsystem P, where T and T' are trace elements representing states of P. In the case of the Rubik's cube, we let state be distinguished by the colours shown on each face, and let P represent a subsystem consisting of certain coordinates on the surface of the cube in which the user is interested. We may now examine the rotations which change the colour of these coordinates in the ordering as defined by t. Note that t does not specify to which colour the positions change, merely that they *do* change in a certain order, starting from the colours specified in R and finishing with the colours specified in R'. In particular, we may be interested in the pre-cocartesian lifting of t, which is the underlying morphism $r : R \to R'$, which restricts to t, and which is comprised of the fewest possible state-changes. With composition defined as concatenation, this means that for any other r' which restricts to t, r' consists of the the statechanges prescribed by r, followed by a number of state-changes which do not affect any component of P. The requirement of uniqueness then follows from our definition of equality. For the Rubik's cube, this corresponds to the fewest rotations which allow us to observe these changes to the face.

Clearly, r then represents the least amount of work the system has to do in order to achieve a simulation path t for the subsystem P. Not all morphisms in every subsystem will have pre-cocartesian liftings, and the existence of these is naturally dependent on the definition of morphisms. The most restrictive definition (above) admits of very few systems for which a relatively small subsystem may have a pre-cocartesian lifting, because it is rare for a small subsystem P to 'control' the possible morphisms r' in the system to such an extent. Such a definition is primarily used when we want to examine two subsystems P, P' which together uniquely define the morphisms of a system. Such an approach is similar to the idea of database view complement [3] and is of interest to Rosetta users because it provides a mathematical analog to the notion that the different perspectives provided by facets in fact *define* a system. In the future, we will introduce new abstraction mechanisms to distinguish objects within the category L. For example, we can consider a state

as being defined by a functor $R : C_V \to \mathbf{Set}$, restricted to elements which are not of type `transition-number`. That is, the number of transitions we take to get to a particular state does not figure in distinguishing the state. Alternatively, we can use a method of state-space reduction, which amalgamates states for which the values of all 'accessible' variables, including those in the subsequent states, are the same. This is used to analyse systems in which information from some states is lost, and hence it becomes no longer possible to tell some states apart. We will also contrast the different definitions of morphism. These incorporate changes such as allowing equality of morphisms to be relaxed to something less restrictive than syntactic equality. For example, we permit the sequence of intermediate $[\bar{R}_i]$ which make up a morphism to be re-ordered without this resulting in a new morphism. This lets us dissociate those changes in a morphism which have no impact upon each other, so that it no longer matters in which order the system performs them. Another proposed change is to compare morphisms which are defined by the components which must change state together — as defined here — with morphisms which are defined by the actual states which the system passes through during the course of the associated progression.

8 Discussion

The advantage of a framework based around the diagram in Figure 6 is that, given an inconsistent system, we can identify why these inconsistencies occur without analysis of the code itself. For example, if there is no functor $Y : \bar{C}_V \to \mathbf{Set}$ in Figure 6 such that the diagram commutes, then the problem lies in the interaction of the facets, rather than the individual facets themselves. Such an error cannot be found by testing the components in isolation, as it only occurs when the facets are placed in an environment which implements these conflicting axioms. If R does not preserve limits and colimits, then the constraints imposed on one facet are mutually exclusive and this can be detected by testing in isolation. We may also use this when trying to assess the suitability of components for different environments.

Being able to formulate questions of consistency within a category theoretical framework allows us to use established category theory tools, such as reasoning about the existence of functors, to address these issues. In addition, the framework here permits us to reason about traces and minimal paths by means of the category theoretical concepts of pre-cocartesian and cocartesian liftings. We can therefore identify certain state-changes which *must* be per-

formed in order to produce a given observation, a property particularly useful when specifying systems with critical behaviour such as security or alarm systems. They also serve as an indication of how much work a system must do in order to produce a specified behaviour, and as such indicate the efficiency of any particular implementation.

9 Conclusion

We have produced a semantics based on category theory for the study of Rosetta and other similar specification languages such as Z and VHDL. By modelling a system as a category, we have abstracted away from individual language issues. By adapting the view update problem, we have enabled a user to analyse components both individually and within a chosen environment, as well as study individual states and the progression between them. Our approach has also made it possible to study inconsistent systems and to identify the source of the inconsistency. In addition, the use of precocartesian and cocartesian morphisms within the same category theory framework has enabled us to compare the choices available in a non-deterministic system.

Further work will consider the possibility of defining degrees of correctness of systems, where certain inconsistencies may be tolerated or circumvented. We will also provide further definitions of morphisms between states, which will enable a user to examine a dynamic system from several different perspectives. The application of fibrations [9] is a further extension which allows a user to deduce the behaviour of an unknown system based simply on the behaviour defined by a certain perspective or subsystem. By using a common framework we have ensured that the results from these different types of analysis can all be expressed using the same vocabulary, and that any relationships between them will be immediately apparent.

References

[1] Alexander, P., Kong, C., Ashenden, P., Menon, C. and Barton, D. “Rosetta Usage and Semantics Guide”, URL: http://www.sldl.org, 2003.

[2] Ashenden, A., Peterson, G. and Teegarden, D. “The System Designer’s Guide to VHDL-AMS”, Morgan Kaufmann Publishers, 2002.

[3] Bancilhon, F. and Spyratos, N. “Update Semantics of Relational Views”, ACM Trans. Database Syst. 6, 1981.

[4] Date, C. J. “An Introduction to Database Systems”, Addison Wesley, 1981.

[5] Easterbrook, S. and Nuseibeh, B. "Using ViewPoints for Inconsistency Management", Software Engineering Journal, Vol. 11, 1996.

[6] Goguen, Joseph and Malcolm, Grant. "A Hidden Agenda", Theoretical Computer Science Vol 245, 2000.

[7] Goguen, Joseph and Burstall, Rod. "Institutions: Abstract Model Theory for Specification and Programming", Journal of the ACM, 39, No. 1, Jan, 1992.

[8] van Horebeek, I. and Lewi, J. "Algebraic Specifications in Software Engineering", Springer-Verlag, 1989.

[9] Jacobs, B. Categorical Logic and Type Theory, Elsevier Science, 1999.

[10] Johnson, Michael and Rosebrugh, Robert. "Update Algorithms for the Sketch Data Model",Proceedings of the Fifth International Conference on Computer Supported Cooperative Work in Design, 2001.

[11] Johnson, Michael and Rosebrugh, Robert. "View Updatability Based on the Models of a Formal Specification", LNCS 2021, 2001.

[12] Johnson, Michael, Rosebrugh, Robert and Dampney, C.N.G. "View updates in a semantic data modelling paradigm", Proceedings of the twelfth Australasian Database Conference ADC2001, 2001.

[13] Johnson, Michael and Rosebrugh, Robert. "Universal View Updatability", Unpublished, available at URL: http://www.cs.mq.edu.au/~mike/pub2000.html, 2001.

[14] Kong, C., Alexander, P. and Menon, C. "Defining a Formal Coalgebraic Semantics for the Rosetta Specification Language", JUCS Vol 9, 2003.

[15] Maclane, S. "Categories for the Working Mathematician", 2nd ed. Springer-Verlag, 1997.

[16] Menon, C., Lakos, C. and Kong, C. "Towards a Semantic Basis for Rosetta", Proceedings of the 27th Australasian Computer Science Conference, 2004.

[17] Rubik's Cube Solution website, URL: http://lar5.com/cube/

[18] Schwanke, R. and Kaiser, G. "Living with Inconsistency in Large Systems", Proceedings of the International Workshop on Software Version and Configuration Control, 1988.

[19] Valmari, A. "The State Explosion Problem", Lectures in Petri Nets 1: Basic Models, Lecture Notes in Computer Science Vol. 1491, Springer Verlag, 1998.