Journal of Universal Computer Science, vol. 9, no. 11 (2003), 1322-1349
submitted: 12/6/03, accepted: 5/9/03, appeared: 28/11/03 © *J.UCS*

Defining a Formal Coalgebraic Semantics for The Rosetta Specification Language

Cindy Kong and Perry Alexander
(The University of Kansas
Dept of Electrical Engineering and Computer Science
Information and Telecommunication Technology Center
2335 Irving Hill Road
Lawrence, Kansas 66045
{ckong,alex}@ittc.ku.edu)

Catherine Menon
(Department of Computer Science
The University of Adelaide
SA 5005, Australia
menon@cs.adelaide.edu.au)

Abstract: Rosetta is a systems level design language that allows algebraic specification of systems through facets. The usual approach to formally describe a specification is to define an algebra that satisfies the specification. Although it is possible to formally describe Rosetta facets with the use of algebras, we choose to use the dual of algebra, i.e. coalgebra, to do so. Coalgebras are particularly suited for describing state-based systems. This makes formally defining state-based Rosetta quite straightforward. For non-state-based Rosetta, the formalization is not as direct, but can still be done with coalgebras by focusing on the behaviors of systems specified. We use denotational semantics to map Rosetta syntactic constructs into a language understood by the coalgebras.
Key Words: coalgebra, state-based, system behavior, algebraic specification, system level design language, formal semantics, denotational semantics
Category: F.1.1, F.3.2, F.4.3

1 Introduction

An important part in the development of a new language is to formally define what the language denotes. This is especially true for specification languages. For most algebraic specification languages, the formalization of the language consists of describing the algebras that satisfy a specification [Ehrig and Mahr 1985]. A similar approach is used in the formal definition of the Rosetta system level design language. However, instead of defining algebras, coalgebras, the duals of algebras, are defined. Coalgebras are particularly suited for describing state-based systems, consequently for describing state-based domains in Rosetta. However, Rosetta is not restricted to state-based computation. Indeed, one of the major benefits of Rosetta is its flexibility in allowing different models of computation, such as trace-based, event-based, graph-based and the like. The semantics of systems from these different models of computation can still be expressed with coalgebras that represent their behaviors.

The link between coalgebras and transition systems is clearly defined by Alexander Kurz [Kurz 2001]. He first defines a theory of systems describing relations between different systems, and then demonstrates how certain systems give rise to coalgebras. Our use of coalgebras in formally defining Rosetta specifications is founded on his work. Other insights to the relation between state-based dynamical systems and coalgebras are given in several papers. Jacobs and Rutten [Jacobs and Rutten 1997] cite several of them in their tutorial on (co)algebras and (co)induction. Of special interest to us, that tutorial describes how a function can be coinductively defined over coalgebras. We use the same approach to define extension relations between Rosetta units of specification.

Rosetta [Alexander et al. 1999, Alexander and Kong 2001] is a systems level design language that uses facets as units of specification. A facet is a signature, consisting of operators, functions and variables, as well as a set of terms that define constraints over the signature. It is used to specify a view of a system or component in terms of some model of computation. The semantics of a model of computation is defined in a special facet, called domain. A facet is said to extend a domain when it consistently uses, adds to or constrains the definitions of that domain. The facet signature is also augmented by the domain signature. A domain can be an extension of another one similarly to a facet. The formal definition of facets and domains consists of describing coalgebras for them. We use denotational functions to map Rosetta syntax to coalgebra semantics. Then, extensions are defined by coinduction over the coalgebraic structures of facets and domains.

In the next sections, we elaborate on the formalization of the semantics of Rosetta facets and domains. We first present some coalgebraic definitions in the background section, derived from both Kurz [Kurz 2001] and Jacobs and Rutten's [Jacobs and Rutten 1997] papers. We then provide a general approach to the denotation of Rosetta facets to coalgebras. The denotation is called α and is divided into two parts. The first part involves representing a facet as a coalgebra and the second consists of denoting Rosetta terms into a language understood by the coalgebra. We then describe the formalization of some specific Rosetta domains and relations into coalgebraic structures. Following this, an example of formally defining a Rosetta domain and facet is given. The next section then describes how commuting diagrams are used to define special functions across domains that are then used to define interactions. The conclusion section finally completes this paper with a description of future work.

2 Coalgebraic Background

This section provides an overview of coalgebras and the use of coalgebras in the semantics of systems. The definitions are summarized from Kurz's lecture notes [Kurz 2001] and Jacobs and Rutten's tutorial [Jacobs and Rutten 1997] in Section 2.1 and Section 2.2 respectively. Kurz defines a theory of systems and describes the semantics of some systems as coalgebras. Jacobs and Rutten define coalgebras for functors and uses special relationships between coalgebras to coinductively define functions. Our definition of Rosetta semantics uses Kurz, and Jacobs and Rutten's work as a basis.

2.1 Modeling systems by coalgebras

2.1.1 Theory of systems

A theory of systems describes the relation of systems and their behaviors in terms of a given interface. Systems are reactive and communicate with other systems and the environment through interfaces. A system is considered to be a set of states X and a transition-function ξ describing for every state $x \in X$ the effect $\xi(x)$ of taking an *observable transition* in state x. A system is thus a function: $X \xrightarrow{\xi} \Sigma X$, where the notation ΣX indicates the set of possible outcomes of taking a transition. Σ is called the type or **signature**, X is called the **carrier** or set of states of the system, and ξ is called the **structure** or transition-function of the system. A **process** is a system together with a given state (usually the initial state) and is denoted by $((X, \xi), x_0)$ or shorter (X, ξ, x_0). A process (X, ξ, x_0) is called a **stream** when the associate system can output elements of a fixed set A forever. Such a system can be represented by a function:$X \xrightarrow{\xi} A \times X$.

A signature Σ for systems is an operation mapping a set (of states) to a set ΣX containing the possible effects of an observable transition. As an interface is to specify the "observable effect" of a transition, Σ itself provides an appropriate notion of interface. The **behavior of the process** $((X, \xi), x_0)$ is given by:

$$Beh(x_0) = (a_0, a_1, a_2 \ldots)$$

This type of behavior thus describes what can be observed of the system ($X \xrightarrow{\xi} A \times X$) when it produces an infinite list $(x_0, (a_0, x_1), (a_1, x_2), \ldots)$, starting from x_0 and taking a transition $\xi(x_0) = (a_0, x_1)$ then continuing with $\xi(x_1) = (a_1, x_2)$ and so on.

Given a process is state dependent, a system has as many processes as states and therefore has a behavior assigned to everyone of its states. The **behavior of a system** is the set of all these behaviors. A fundamental observation is that *the behavior of a system is itself a system*, i.e. it can be described as a set of states and a transition-function. Let (X, ξ) be a system and $Beh(X) = \{Beh(x) : x \in X\}$ the set of all behaviors of X. For $Beh(X)$ to be considered as a system, we have to exhibit a transition-function $\beta : Beh(X) \to A \times Beh(X)$. β has to map an infinite list $l = (a_0, a_1, a_2, \ldots)$ into $A \times Beh(X)$. An obvious candidate is:

$$\beta : Beh(X) \to A \times Beh(X)$$
$$(a_0, a_1, a_2, \ldots) \mapsto \langle a_0, (a_1, a_2, \ldots)\rangle$$

Note that the behavior of some $l \in Beh(X)$ is l and that the behavior of system $(Beh(X), \beta)$ is $(Beh(X), \beta)$.

The interest in a general theory of systems lies in the relationships between different systems or in structural properties of collections of systems. System relationships are investigated by using structure preserving mappings between systems. Given $head(x)$ represents the first value a and $tail(x)$ the remainder x' of a stream (X, ξ, x) with $\xi(x) = (a, x')$, a homomorphism, or morphism for short, between two systems $X \xrightarrow{\xi} A \times X$ and $X' \xrightarrow{\xi'} A \times X'$ is a function $f : X \to X'$ such that

$$head(f(x)) = head(x) \text{ and } tail(f(x)) = f(tail(x))$$

The precise definition of behavior of a system $X \to A \times X$ at state $x_0 \in X$ is then defined as $Beh(x_0) = (head(tail^n(x_0)))_{n \in \mathbb{N}}$ where $tail^n$ is defined inductively via $tail^0(x) = x$, $tail^{n+1}(x) = tail(tail^n(x))$. Behaviors are invariant under morphism and $Beh : X \to Beh(X)$ is the unique morphism $(X, \xi) \to (Beh(X), \beta)$. Therefore, two states have the same behavior if and only if these states are identified by some morphisms.

Much of the power of a general theory of systems comes from the observation that *all behaviors of all systems constitute themselves a system*. For any process (X, ξ, x), the behavior is an infinite list $(a_i)_{i \in \mathbb{N}}$. The set of all behaviors of all processes is thus given by $A^{\mathbb{N}} = \{f : \mathbb{N} \to A\} = \{(a_i)_{i \in \mathbb{N}}, a_i \in A\}$. As for the behavior of a process, this set of all behaviors of all processes can be made into a system with transition structure:

$$\zeta : A^{\mathbb{N}} \to A \times A^{\mathbb{N}} \tag{1}$$
$$(a_0, a_1, a_2, \ldots) \mapsto \langle a_0, (a_1, a_2, \ldots)\rangle$$

Since the mapping from a system to its behavior is a morphism, we know that, for any system, there must exist a morphism into the system of all behaviors (namely the one mapping each process to its behavior). And, since morphisms preserve behaviors, for any system, there can be at most one morphism into the system of all behaviors. Thus, the system of all behaviors is a final system. A system (Z, ζ) is called *final* (or *terminal*) if and only if for all systems (X, ξ) there is a unique morphism $(X, \xi) \to (Z, \zeta)$.

Two processes/systems are behaviorally equivalent if and only if they have the same behavior. Formally, given two systems, (X, ξ) and (X', ξ'), and Beh and Beh' the two corresponding unique morphisms into the final system.

1. Two processes (X, ξ, x) and (X', ξ', x') are behaviorally equivalent iff $Beh(x) = Beh'(x')$.

Σ	ΣX	Process	System
1	1	stop	$X \xrightarrow{\xi} 1$
A	A	output $a \in A$ once	$X \xrightarrow{\xi} A$
Id	X	metronome (running forever)	$X \xrightarrow{\xi} X$
$A \times -$	$A \times X$	stream over A	$X \xrightarrow{\xi} A \times X$
$A \times X - +1$	$A \times X + 1$	finite or infinite list over A	$X \xrightarrow{\xi} A \times X + 1$

Table 1: Some systems and their signatures

2. Two systems (X, ξ) and (X', ξ') are behaviorally equivalent iff $Beh(X) = Beh'(X')$.

$R \subset X \times X'$ is a bisimulation over two systems of streams (X, ξ) and (X', ξ') iff

$$x\ R\ x' \Rightarrow head(x) = head(x') \text{ and } x\ R\ x' \Rightarrow tail(x)\ R\ tail(x')$$

In other words, $x\ R\ x'$ implies that a transition $x \mapsto \langle head(x), tail(x) \rangle$ can be simulated by a transition $x' \mapsto \langle head(x'), tail(x') \rangle$ and vice versa. Two processes are behaviorally equivalent if and only if they are bisimilar.

Since one is usually interested in processes only up to behavioral equivalence, it is therefore sensible to consider behavioral equivalence as equality on processes. In the final system, two processes are behaviorally equivalent iff they are equal. The principle of **definition by coinduction** can then be used. Since for any system $X \xrightarrow{\xi} \Sigma X$ there is a *unique morphism* into the final system (Z, ζ), we can define a function $f : X \to Z$ just by giving an appropriate structure:

$$\text{for all } X \xrightarrow{\xi} \Sigma X \text{ there is a unique morphism } (X, \xi) \xrightarrow{f} (Z, \zeta).$$

We say that function f is defined by coinduction if it arises in such a way from a $\xi : X \to \Sigma X$.

Systems with inputs are modeled as $X \times I \to X$. However, as mentioned previously, a system is a function $X \xrightarrow{\xi} \Sigma X$, i.e. of the kind $(X \to \ldots)$ and not $(\ldots \to X)$. Currying is therefore used to write functions representing systems with inputs in the correct form. Given $f : X \times I \to X$, $f(x, \_)$ is a function $I \to X$ for each $x \in X$. It follows that $f(\_, \_)$ is a function from X to the functions $I \to X$. Therefore, given sets I and X, and denoting X^I to be the set of functions from $I \to X$, systems with inputs $(X \times I \to X)$ can now be written as $(X \to X^I)$.

Table 1 describes the signatures for some processes. `1` denotes a one-element set and `Id` denotes the identity operator.

Σ	ΣX	Σf
C	C	$id_c : C \to C$
Id	X	f
$(-)^C$	X^C	$f^C : X^C \to Y^C$ $g \mapsto f \circ g$

Table 2: System signatures that are functors

2.1.2 Coalgebra of systems

The theory of systems is almost uniform in all signatures, except for the notion of morphism that has to be created separately for each new signature. The idea is thus to define the signature such that it includes, in a natural way, the right notion of morphism. This is done by requiring the signature to be a functor.

Given a category χ, called the base category, and a functor $\Sigma : \chi \to \chi$, a **Σ-coalgebra** (X, ξ) is given by an arrow $\xi : X \to \Sigma X$ in χ. A morphism between two coalgebras $f : (X, \xi) \to (X', \xi')$ is an arrow f in χ such that $\xi' \circ f = \Sigma f \circ \xi$:

$$\begin{tikzcd} X \arrow[r, "\xi"] \arrow[d, "f"'] & \Sigma X \arrow[d, "\Sigma f"] \\ X' \arrow[r, "\xi'"'] & \Sigma X' \end{tikzcd}$$

Assume $\chi = Set$ where Set is the category of sets, i.e. the objects of the category are sets. Some signatures can be extended to functors as shown in Table 2 (let $C \in Set$ and $f : X \to Y \in Set$). id_C denotes the identity map on C and X^C is function space.

As the signatures of systems in Table 1 give rise to functors, these systems can be represented as coalgebras. In particular, the general theory of systems outlined previously is now uniformly available to all categories of coalgebras over sets.

2.2 Coalgebras of functors

A $functor$ is an operator on sets that also act on functions between sets while preserving identity functions and composition functions. A “polynomial” functor T is a functor built up with constants, identity functors, products and coproducts and also (finite) powersets. For example, $T(X) = X + (C \times X)$ where C is a constant set and X a set. For a functor T, a *coalgebra* (or a T-coalgebra) is a pair (U, c) consisting of a set U and a function $c : U \to T(U)$. The set U is called the *carrier* and the function c is the *structure* or *operation* of the coalgebra (U, c). The carrier set is also called the *state space*.

A *homomorphism of coalgebras* from a T-coalgebra $U_1 \overset{c_1}{\to} T(U_1)$ to another T-coalgebra $U_2 \overset{c_2}{\to} T(U_2)$ consists of a function $f : U_1 \to U_2$ between the carrier sets which commutes with the operations: $c_2 \circ f = T(f) \circ c_1$. A *final coalgebra* $d : W \to T(W)$ is a coalgebra such that for every coalgebra $c : U \to T(U)$ there is a unique map of coalgebras $(U, c) \to (W, d)$.[1]

A map can be defined with the use of a finality diagram (Figure 1). The map "and-so-forth" applies the "next step" operations repeatedly to the "base step". The technique for defining a function $f : V \to U$ by finality is thus: describe the direct observations together with the single next steps of f as a coalgebra structure on V. The function f then arises by repetition.

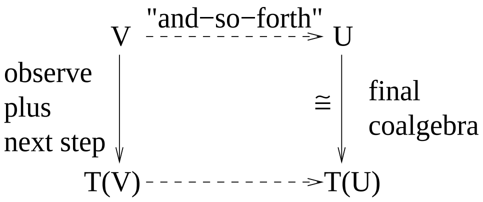

Figure 1: Coinductive definition of a function

Example 1. Co-inductive definition of the *merge* function.

As an example, the function *merge* is coinductively defined. Given a functor $T(X) = A \times X$ where A is a fixed set, its final coalgebra is the set $A^{\mathbb{N}}$ of infinite lists of elements from A, with coalgebra structure

$$\langle head, tail \rangle : A^{\mathbb{N}} \to A \times A^{\mathbb{N}}$$

given by

$$head(\alpha) = \alpha(0) \text{ and } tail(\alpha) = \lambda x.\alpha(x + 1)$$

For an arbitrary coalgebra $\langle value, next \rangle : U \to A \times U$ there is a unique homomorphism of coalgebras $f : U \to A^{\mathbb{N}}$; it is given for $u \in U$ and $n \in \mathbb{N}$ by

$$f(u)(n) = value(next^{(n)}(u))$$

What can be observed about an element $u \in U$ is an infinite list of elements of A arising as $value(u), value(next(u)), value(next(next(u))), \ldots$ This observable behavior of u is precisely the outcome $f(u) \in A^{\mathbb{N}}$ at u of the unique map f to the final coalgebra. Hence the elements of the final coalgebra give the observable behavior. This is typical for final coalgebras.

Once it is known that $A^{\mathbb{N}}$ carries a final coalgebra structure, this finality can be used to define functions into $A^{\mathbb{N}}$. For instance, the function *merge* :

[1] Note the similarity with the definitions in Kurz's theory of systems (Section 2.1.1).

$A^{\mathbb{N}} \times A^{\mathbb{N}} \rightarrow A^{\mathbb{N}}$ which merges two infinite lists into a single one arises as a unique function to the final coalgebra $A^{\mathbb{N}}$ in:[2]

$$\begin{tikzcd}
A^{N} \times A^{N} \arrow[r, "\text{merge}"] \arrow[d, swap, "\lambda(\alpha,\beta).\ (\text{head}(\alpha),(\beta,\ \text{tail}(\alpha)))"] & A^{N} \arrow[d, "\cong\ \langle \text{head, tail} \rangle"] \\
A \times (A^{N} \times A^{N}) \arrow[r, "\text{id} \times \text{merge}"] & A \times A^{N}
\end{tikzcd}$$

3 Formal denotation of Rosetta specifications into coalgebras

3.1 Formal denotation of facets

We call the formal denotation of a Rosetta specification into a coalgebra an α-denotation. In this section, we define the function for the α-denotation of a facet. Note that by $facet$, we intend a consistent facet. Although we do not prove the consistency of any facet described in this work, we assume that it is consistent. It automatically follows that if a facet is inconsistent, then the coalgebra denotation and the theory presented in this paper do not hold for that facet.

A facet, F, can be considered as a 4-tuple:

$$\langle l,\ O,\ D,\ T\ \rangle$$

where l is the label, O is composed of all the parameters and variables (i.e. all the observers of the facet), D is the domain that the facet extends, and T contains the terms of the facet. Whatever the domain the facet extends, the behavior is defined by observing the parameters and variables of the facet, O, as well as the parameters and variables defined in the extended domain O_D.[3]

The coalgebraic structure of the facet can be defined by using its domain coalgebra. Given the coalgebraic structure of the domain is $X^{\ltimes} \xrightarrow{\zeta} X \times X^{\ltimes}$, where $\ltimes$ is some number, then the coalgebra of the facet (left of diagram) is given such that the following diagram (Figure 2) commutes. The domain coalgebra describes the behaviors of all facets extending D. Consequently, the carrier set X represents a set of tuples of different sizes, with each tuple containing at least the variables defined in D, i.e. each element of X will be a tuple containing O_D

[2] Note that the coalgebra on the left expresses the direct observation after a merge, together with the next state (about which a next direct observation is made).

[3] The parameters and variables of any parent domain of the extended domain are also observed at the facet level, i.e. observation of parameters and variables is recursive on the extension of domains.

and variables defined in facets extending D. In this diagram the *extn* morphism only involves the subset of X consisting of tuples of $\langle O, O_D \rangle$. As *extn* describes the map from the behaviors of facet F to the system of all behaviors as defined by D, it is similar to an identity function. It will map a specific tuple to the same tuple and a sequence of tuples to the same sequence of tuples. Although *extn* has the properties of a coalgebraic morphism, it is not of much interest as an "identity" function. However, it is the unique map between each facet coalgebra and its domain coalgebra, thus resulting in the domain coalgebra being final.

$$
\begin{tikzcd}
\text{facet F} & \text{domain D} & \text{abstracted D'} \\
(O \times O_D)^{\alpha} \arrow[r, "\text{extn}"] \arrow[d, "\xi"] & X^{\alpha} \arrow[r, "\text{map(f)}"] \arrow[d, "\zeta"] & O_D^{\alpha} \arrow[d, "\zeta'"] \\
O \times O_D \times (O \times O_D)^{\alpha} \arrow[r, "\text{id} \times \text{extn}"] & X \times X^{\alpha} \arrow[r, "\text{f} \times \text{map(f)}"] & O_D \times O_D^{\alpha}
\end{tikzcd}
$$

$$
where \begin{cases} f : O \times O_D \rightarrow O_D \text{ i.e. describes an abstraction of variables)} \\ \xi : (O \times O_D)^{\ltimes} \rightarrow O \times O_D \times (O \times O_D)^{\ltimes} \text{ such that } T \text{ holds} \end{cases}
$$

Figure 2: Coalgebra of a facet

As all facets extending the same domain D observe the variables defined in that domain, it is of more interest to see how facets affect these variables. The move from *domain* D to *abstraced* D' in Figure 2 (represented by $map(f)$ and $f \times map(f)$) can be considered as restricting the observed behavior to the variables of the domain. In some ways, this can be viewed as an `abstraction` of the observed behavior of a facet. From here onward, the behavior forming the facet coalgebra consists of observing all variables (including parameters) of the facet and of the domain the facet extends. The behavior forming the domain coalgebra consists of observing only the variables from the domain.[4]

3.2 Formal denotation of Rosetta terms into λ-expressions of coalgebras

In the previous section, the function ξ of the coalgebraic structure of the facet depends on the terms defined in the facet. It is therefore necessary to under-

[4] It is understood that in truth, all variables observed in a facet are also present in the domain coalgebra. To analyze the effects of extension, we choose to look at the domain variables isolated from facet variables.

stand what a Rosetta term denotes. We thus provide denotational functions for such terms. We use the phrase "denotation" in the sense of *The Scott-Strachey Approach to Programming Language Theory* [Stoy 1977]. We provide "semantic valuation functions" that map syntactic constructs in the program to the abstract values that they denote in the λ-language understood by the coalgebra. However, because the semantics of some syntactic constructs is dependent on domains, the valuation functions may vary from one domain to another. We therefore start by defining three basic valuation functions that are used in the prelude of the language (defines the *static* domain). We then modify these functions to make them match the semantics defined within each domain.

3.2.1 Formal denotation of prelude terms

We define three basic semantic valuation functions: one for expression, one for constants and literals, and one for operators. The expression valuation function is defined as $E[\![\varepsilon]\!] : Environment \to Values$ such that, given an environment $envt$, it evaluates an expression to its value ($[\![]\!]$ indicates semantic brackets). An $envt$ contains known mappings between identifiers and values as well as between functions and values/body of these functions. E can also be applied to undefined functions in some cases. Given values of an undefined function are known in $envt$ for specific tuples of parameter values, the value of such a function applied to a particular tuple of parameters can be obtained. Something similar to pattern matching can be used to determine the function value. $Values$ refer to any kind of value and is the $universal$[5] type in Rosetta. $Universal$ is the set of all values, including function values (i.e. lambda expressions). Function $V : Constants \to Values$ is the valuation function for constants/literals. Valuation functions for operators are in the form $O[\![\Omega]\!] : universal \to universal \to universal$, where Ω is an operator.

Following are some case by case definitions of the valuation functions as well as some examples of their applications.

- $E[\![\xi]\!]\ (envt) \equiv (envt\_value\ (\xi))$
- $E[\![v]\!]\ (envt) \equiv V[\![v]\!]$
- $E[\![\varepsilon\Omega\varepsilon']\!]\ (envt) \equiv O[\![\Omega]\!]\langle E[\![\varepsilon]\!]\ (envt), E[\![\varepsilon']\!]\ (envt)\rangle$
- $O[\![=]\!] \equiv \lambda\langle v1,\ v2\rangle.if\ v1 = v2\ then\ True\ else\ False$
  $: universal \to universal \to bool$

[5] We use $Values$ and $universal$ interchangeably.

$$where \begin{cases} \varepsilon \text{ is an expression} \\ envt \text{ represents the } environment \\ \Omega \text{ is a binary operator} \\ \lambda\langle parameters\_lmbd\_fnctn\rangle.body\_lmbd\_fnctn : type\_lmbd\_fnctn \\ \quad\text{ is a lambda expression of type } type\_lmbd\_fnctn \\ \xi \text{ represents identifiers} \\ v \text{ represents constants} \\ (evnt\_value\ (x)) \text{ is a shortcut for saying the value of } x \text{ in the} \\ \quad \text{environment} \end{cases}$$

The interpretation of a term may result in a specific *true* or *false* value, or in an equation with unknowns. If the values of all the variable identifiers in the denoted equation are known, then the equation can be evaluated to a *true* or *false* value. However, if one or more variable identifiers are unknown (an undefined function with known parameters can be considered as an unknown identifier), then, it may not always be possible to evaluate the equation to a specific boolean value. In this case, a system of equations will be obtained and a constraint solver may be used to try to solve the unknowns. It is always assumed that all equations with unknowns are evaluated to *true* for consistency of a facet.

The valuation function for terms is $T : Terms \rightarrow Environment \rightarrow boolFnc$, where $boolFnc$ is a function $Vars \rightarrow Boolean$ with $Vars$ representing any number of variables, i.e. a term can be evaluated to a boolean value, or to a boolean function of one parameter, or to a boolean function of two parameters, and so on. The resulting function of evaluating a term depends on the term and on the environment. For example, an evaluation results in a boolean function of one parameter when an item in the term is undefined, i.e. is of unknown value.

Some simple examples of term evaluations are given below. Assume that the environment $envt$ contains the following mappings:

- $x \equiv 2$
- $myFnc \equiv \lambda x : int\ .\ x\ +\ 1$

then, the following terms are evaluated as follows (not all the steps of the evaluations are shown):

- $T[\![x = 2]\!]\ (envt) \equiv O[\![=]\!](E[\![x]\!]\ (envt))(V[\![2]\!]) \equiv$
  $if\ (2 = 2)\ then\ True\ else\ False \equiv True$
- $T[\![(myFnc\ 1) = 2]\!]\ (envt) \equiv O[\![=]\!]\ ((\lambda x : int\ .\ x\ +\ 1)\ (1),\ 2) \equiv True$
- $T[\![(undefFnc\ x) = 2]\!]\ (envt) \equiv (undefFnc\ (2))\ =\ 2$
  where $(undefFnc\ (2))\ =\ 2$ is a boolean equation with one unknown

3.2.2 Formal denotation of state-based terms

We use the notion of a unit of semantics to describe a unifying semantic domain. A unit of semantics provides a domain of discourse with a vocabulary and a semantics for that vocabulary. The underlying semantics defines the rules that provide meaning to the vocabulary. A model of computation can often be expressed using different representation. For example, a Kahn process is naturally represented by functions over streams of values. However, it can also be represented as a state machine that simulates its behavior [Burch et al. 2001]. The vocabulary contains the objects that are needed to describe a model of computation. The semantics describes what they mean and the rules that govern how they interact.

As shown in Figure 3, we originally propose two units of semantics, the *signal_based* domain based on the Tagged Signal Model [Lee 1998], and the *state_based* domain for state-based models of computation. Signal-based domains specify models with the use of events and signals, with a signal being a set of events and an event a pair of tag and value. The underlying semantics defines what the values and tags are, and describes the meaning of a process and the rules that govern how processes behave (firing rules and communication protocols). In the case of state-based, models are specified with the notion of states and state transitions. In this case, the semantics describes states with respect to the inputs, outputs and transition functions. Due to the difference in semantics for these two domains, each has slightly different denotational valuation functions. In this paper, we only show the denotation of state-based terms.

We identify the vocabulary of a state based model to consist of a set of states, K (finite or infinite, denumerable or nondenumerable), a finite set of inputs and outputs, Γ, and a transition relation that given a state returns a next state or a set of next states, $K \times \Gamma \rightarrow K$. The semantics of a state based model can be associated with the transition relation as the latter provides the rule describing how the state transformations occur.

The definition of this semantics is achieved in two steps. The first step is to define what a state denotes. Allison [Allison 1986] defines a state to be the values of variable identifiers in the sequential execution of a program. The set of states, $S = \{Var \rightarrow Values\}$, thus represents the set or data-type of functions from identifiers (Var) to values ($Values$). A particular state $\sigma : S$ is a particular function from variables to values. The second step is in denoting a state transformation function. Allison defines a command to denote a relation ($S \rightarrow S$), with the valuation function for commands being $C : Cmd \rightarrow (S \rightarrow S)$. He also describes the notion of expressions along with a function that evaluates the expression in a given state ($E[\![\varepsilon]\!] : S \rightarrow Value$). A command differs from an expression in that it does not have an intrinsic value as does an expression. It also causes a state transformation whereas evaluating an expression does not (assuming no side effects). However, if expressions have side-effects, another expression valuation function is used: $E : Exp \rightarrow S \rightarrow Value \times S$ giving $E[\![\varepsilon]\!] : S \rightarrow Value \times S$.

Vocabulary		Semantics	
Item	**Description**	**Denotation**	**Description**
$States$	Set of states	Set	Set of states
Var	Set of variables	$\{States \rightarrow Values\}$	Set of functions from states to values
σ	Current state	$\sigma \in States$	current state
$v \in Var$	Example of a variable	$States \rightarrow Values$	Particular function
$next$	State transformation relation	$States \rightarrow States$	Transformation function
@	*apply* function for dereferencing labels	$(States \rightarrow Values) \rightarrow States \rightarrow Values$	Similar to *apply* function in Lisp - applies a $v \in Var$ function to state and returns the value of the application

Table 3: Vocabulary and semantics of state_based

Although our state based model is not restricted to defining sequential program execution, we can still mirror its semantics on Allison's definitions. Instead of having a state be a function from variable to value, we take a dual approach, where each variable is a function of a state to a value, i.e. $Var = S \rightarrow Values$ (see Appendix A for demonstration of duality). Another deviation from Allison's work is based on the absence of a *command* in our declarative language. There is no notion of assignment or side-effects in Rosetta. The side-effect of a command has to be explicitly defined such that the change of state is no longer hidden within the command, but made explicit. It is therefore understood that the same identifier in two different states denotes two different instances of a variable.

The vocabulary and semantics associated with the state-based unit of semantics are that of a state machine as shown in Table 3. The valuation functions, T and E, defined in the prelude (Section 3.2.1) are also used here. As mentioned previously, terms are used to define properties over elements of a vocabulary. For example, in sequential execution, with $x := x + 1$, it is understood that there is a state change such that x in the new state is equal to the value of x in the previous state plus one. In Rosetta, there is no sequential execution. To achieve the same idea, we have to clearly express $x@next(s) = x@s+1$ where $x@next(s)$ means x in next state since s is the current state. For this reason, elements of the vocabulary are visible in the language. Following this, the denotational semantic functions can be considered to interpret rather than to evaluate.

We define some of the valuation functions as follows (we use the same notations as in Section 3.2.1):

$$O[\![@]\!] \equiv \lambda\langle fnc,\ s\rangle.fnc(s) : (States \rightarrow universal) \rightarrow States \rightarrow universal$$
$$E[\![y@\sigma]\!]\ (envt) \equiv E[\![y]\!]\ (envt, \sigma) \equiv (envt\_value\ (y))\ (envt\_value\ (\sigma))$$
$$E[\![y@next(\sigma)]\!]\ (envt) \equiv$$
$$(envt\_value\ (y))\ ((envt\_value\ (next))\ (envt\_value\ (\sigma)))$$

Each function $v \in Var$ may not have a complete direct definition. In such cases, the function is either defined as a set of pairs $(States, Values)$ with the pairs listed in *envt* when known, or as an abstract function for which certain conditions must hold (these conditions are defined by the terms of the facet). Due to the way the next state is defined over the transformation of a variable, although the *next* relation may be undefined, the value of a variable in the next state may still be known. For example, if the value of $x@s$ is known, then the value of $x@next(s)$ where $x@next(s) = x@s + 1$ is also known without knowing the exact definition of *next*.

4 Coalgebras of domains

The coalgebras of domains are represented with the use of coalgebraic structures that define the behaviors of systems represented in each domain. Figure 3 provides some basic domains in Rosetta, as well as the "extension" relations[6] that exist between them. As the nodes of the tree represent coalgebras, the arrows go from a less abstract domain to a more abstract one. A domain is said to be more abstract when it defines less constraints. A more abstract domain allows for more observations (cf. Appendix B). The root of the tree is the *null* domain where there is no restriction on observations made. The next domain is the *static* domain where at least constants have to be observed over all behaviors, i.e. all observations will include some items that do not change values whatever the behavior. In the *state_based* domain, at least a variable representing state identifier is observed over the behaviors of systems. These behaviors are obtained by following some transition function. As for any domain, the *state_based* domain also covers all observations that can be made by its extending domains. These extending domains all have observations of a state identifier variable in common. However, where the state variable is of some abstract *States* type in *state_based*, in domains extending the latter, its type can be more concretely defined as long as properties of *States* hold for the new type. For example, in the *discrete_time* domain, state transformation is correlated with time that varies discretely. Therefore, the type, *States*, can be equated to the set of natural numbers, and any observation of the state variable gives a natural number. In the

[6] In Rosetta, domain "extension" means adding items and constraints to an existing domain.

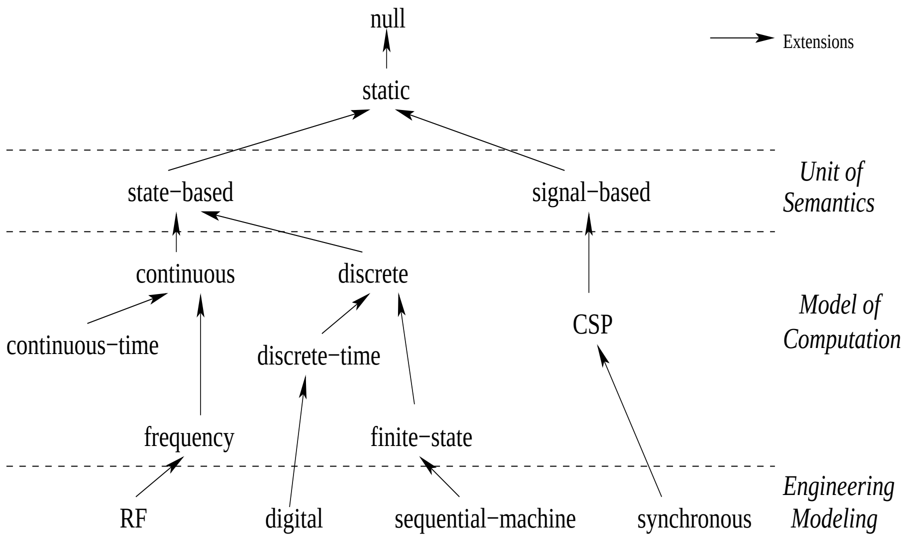

Figure 3: Domain structure

signal_based domain, observations can be made on the events currently seen by a process. The domains extending *signal_based* add constraints to what can be observed.

The relation between two domains, or a domain and a facet can be defined coinductively with the use of the finality of coalgebras [Jacobs and Rutten 1997]. The finality of a domain coalgebra is given by the fact that a domain coalgebra describes all possible behaviors (observable or not) of systems specified within that domain. Assume domain D contains an observable variable $v :: I$. In facet $f1$ extending D, v is defined as evolving according to $v_{n+1} = v_n + 1$. A behavior of the facet (with respect to v) is therefore $(v_n, v_n+1, v_n+2, \ldots)$. In another facet $f2$ also extending D, v evolves as $v_{n+1} = v_n + 2$. The corresponding behavior is thus $(v_n, v_n + 2, v_n + 4, \ldots)$. The system of behaviors according to domain D is $I^{\mathbb{N}} \xrightarrow{\zeta} I \times I^{\mathbb{N}}$. Since $I^{\mathbb{N}} = \{(i_j)_{j\in\mathbb{N}}, i_j \in I\}$, the domain's system of behaviors includes the behaviors of v from both facets (similar to equation 1): $(v_n, v_n + 1, v_n + 2, \ldots) \mapsto \langle v_n, (v_n + 1, v_n + 2, \ldots)\rangle$ and $(v_n, v_n + 2, v_n + 4, \ldots) \mapsto \langle v_n, (v_n + 2, v_n + 4, \ldots)\rangle$. Consequently, there exists a unique morphism from the system of behaviors of each facet to the system of behaviors of the domain (the morphism can be described as a case of application or as "a subset of"), thus making the domain coalgebra final. As the coalgebra of a domain is terminal, we can define the "extension" relations to that domain coinductively. A commuting diagram mapping the coalgebra structure of the extending facet to the coalgebra structure of the extended domain is used. In Figure 4, several commuting diagrams are combined to show how these extensions work.[7] With each domain D we associate a set of coalgebras corresponding to the facets and domains extending D.[8]

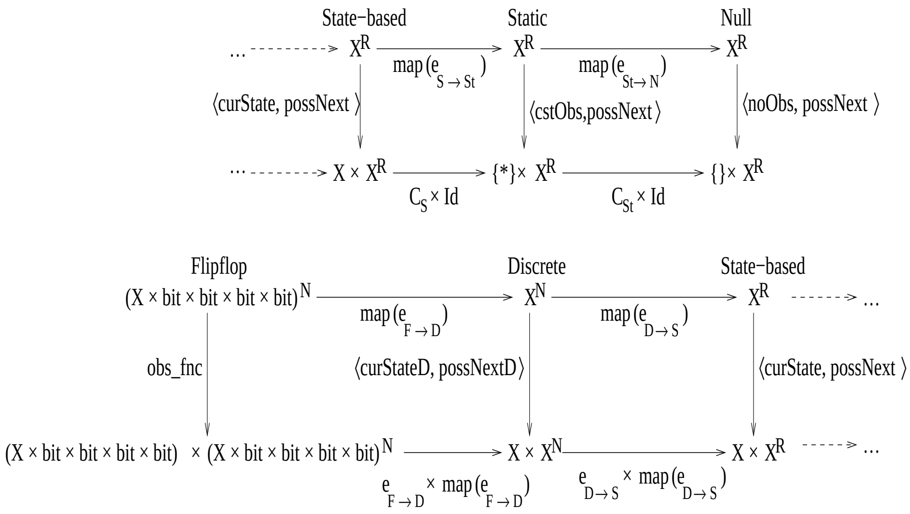

Figure 4: State_based extension

The coalgebraic structure for the *null* domain is given by

$$N = \langle noObs, possNext\rangle :: X^{\mathbb{R}} \rightarrow \{\} \times X^{\mathbb{R}}.$$

The *null* domain is thus represented as one where no restriction is made on observations although its system may run continuously to infinity. While $X^{\mathbb{R}} = \{(x_i)_{i\in\mathbb{R}}, x_i \in X\}$ represents the set of behaviors of something of type X (a behavior can be a continuous sequence of values from X), $\{\}$ indicates that the observations over the behavior are not constrained. Although $\{\} \times S = \{\}$ where S is any set, in the diagram we use the expanded form so that the diagram commutes. Also, we use $\mathbb{R}$ instead of $\mathbb{N}$ for the behaviors to indicate that the sequence can be continuous as well as discrete ($\mathbb{N}$ is a subtype of $\mathbb{R}$).

The *static* domain extends the *null* domain by constraining all the behaviors to at least contain constant observations. As for *null*, the behaviors of systems represented in *static* can be given as sequences of values of X. However, all behaviors need to involve constant observations. The coalgebra structure for the

[7] Note that we work directly on the system of behaviors for each domain. We do not show the relation between the domain system and its behavior system (see Section 5) because we are mainly interested in the coinductive relations between coalgebras. Furthermore, for non state-based domains, we cannot always describe the systems in the form $X \xrightarrow{\xi} \Sigma X$.

[8] In Figure 4, for each of the domain, the coalgebra shown describes the minimum observations allowed for all behaviors of that domain.

static domain is therefore $St = \langle cstObs, possNext \rangle :: X^{\mathbb{R}} \rightarrow \{*\} \times X^{R}$, where $\{*\}$ indicates a singleton set.[9] Whatever the behavior, the constant observation is always the one member of the singleton set.[10] An observation is constant when what is being observed does not change.

The *state_based* domain's coalgebra indicates that elements of a state can be observed, $S = \langle curState, possNext \rangle :: X^R \rightarrow X \times X^R$. At this domain's level, the observations are abstract in that we observe a variable of type, X, but no constraint is put on it.[11] [12] In Figure 4, note how X appears in any coalgebra extending *state_based*. Several domains extend *state_based*, one being *continuous* while another is *discrete*. The coalgebra structure for the *continuous* domain is very similar to that of *state_based*, except for the mapping function, $\langle curStateC, possNextC \rangle :: X^{\mathbb{R}} \rightarrow X \times X^{\mathbb{R}}$.[13] However the coalgebra for *discrete* differs in a characteristic way, the sequences can still be infinite, but are discrete, $D = \langle curStateD, possNextD \rangle :: X^N \rightarrow X \times X^N$.

In the *signal_based* domain, events and their effects can be observed from a specification. The behaviors of a process are thus represented by traces of events as seen by the process. The coalgebra structure is given by:

$$(\{E\})^{\mathbb{R}} \rightarrow \{E\} \times (\{E\})^{\mathbb{R}}.$$

An observation involves a set of events, $\{E\}$, where E indicates an event. $(\{E\})^{\mathbb{R}}$ indicates the sequence of observations where each observation consists of a set of events. As a *signal* is a set of events, $(\{E\})^{\mathbb{R}}$ indicates a sequence of signal values. The extension from the *static* domain to the *signal_based* domain is given below. As can be observed the section of the diagram indicating extension from *null* to *static* is identical to the previous domain extension diagram (Figure 4).

$$\begin{tikzcd}
 & \text{Signal-based} & \text{Static} & \text{Null} \\
\dots \arrow[r, dashed] & (\{E\})^R \arrow[r, "map(e_{Sg \rightarrow St})"'] \arrow[d, "\langle curEvents, futEvents \rangle"'] & X^R \arrow[r, "map(e_{St \rightarrow N})"'] \arrow[d, "\langle cstObs, possNext \rangle"] & X^R \arrow[d, "\langle noObs, possNext \rangle"] \\
\dots \arrow[r, dashed] & \{E\} \times (\{E\})^R \arrow[r, "C_{Sg} \times Id"'] & \{*\} \times X^R \arrow[r, "C_{St} \times Id"'] & \{\} \times X^R
\end{tikzcd}$$

The *csp* domain is an example of a domain extending the *signal_based* do-

[9] Theoretically, all domains/facets extending static should also be observing these constants. However, we do not show them in the coalgebras so as not to clutter the coalgebras.

[10] $\{*\}$ also represents more than one constant values, i.e. it is possible to observe more than one constant values.

[11] This state identifier variable is often referred to as the state as it identifies a state.

[12] Once again, although only X is shown in the *state_based* coalgebra, it is understood that X is just a member of a tuple (cf. Section 3.1).

[13] The difference in the structure function results from additional terms defined in *continuous*. Terms are used to add constraints or properties to variables.

main. In the Tagged Signal Model [Lee 1998], a loop signal is used to model a sequential process. Similarly, in a Rosetta csp-like specification, a control signal is used to keep track of events. A control signal is one that can be read or modified only by the same process (loop back signal). By defining a local signal variable in the *csp* domain in Rosetta, we constrain all facets extending *csp* to have a local signal variable that only they can read.

$$
\begin{tikzcd}
\overset{\text{CSP}}{(E\times\{E\})^{\mathbb{N}}} \arrow[r, "map(e_{Csp\rightarrow Sg})"] \arrow[d, "{\langle\langle ctrlObsC\times curEvtsC\rangle, futEvtsC\rangle}"'] & \overset{\text{Signal-based}}{(\{E\})^{\mathbb{R}}} \arrow[r, dashed] \arrow[d, "{\langle curEvents, futEvents\rangle}"] & \dots \\
E\times\{E\}\times(E\times\{E\})^{\mathbb{N}} \arrow[r, "e_{Csp\rightarrow Sg}\times", "map(e_{Csp\rightarrow Sg})"'] & \{E\}\times(\{E\})^{\mathbb{R}} \arrow[r, dashed] & \dots
\end{tikzcd}
$$

The coalgebra structure for *csp* is given above. Due to the specific identification of a control signal, we make the observation of the events in that signal explicit. Therefore, although the coalgebraic function is still composed of two functions as for other domains, the current observation function is a product of two functions:

$$CSP = \langle(ctrlObsC \times curEvtsC), futEvtsC\rangle ::$$
$$(E \times \{E\})^{\mathbb{R}} \rightarrow (E \times \{E\}) \times (E \times \{E\})^{\mathbb{R}}$$

ctrlObsC provides the observation of the current event from the control signal. *curEvtsC* observes other events that happen at the same time (or within a δ delay) as the event from *ctrlObsC*. After a process reads a specific event from the control signal, it may read further events from other signals, process them and add new events to some signals. *curEvtsC* gives all these other events that are processed as well as created. $futEvtsC$ gives the rest of the sequence of events that describes the behavior of the process.

5 An example in Rosetta

As an example of α-denoting Rosetta, we look at the specification of a discrete flip-flop. We provide the formal semantics, giving the ratiocination behind each step, starting with the *discrete* domain. The Rosetta specification of *discrete* is given below. It expresses that the set of states consists of distinct, countable elements.

```
domain discrete :: state_based is
begin
  d1: exists (fnc::<*(st::States)::natural*> |
```

```
        forall(s1,s2::States|
          (s1 /= s2) implies (fnc(s1) /= fnc(s2))))
end domain discrete;
```

A facet represents an aspect of a system (X, ξ) and therefore can be expressed in the form $X \xrightarrow{\xi} \Sigma X$ where Σ is called signature as in Section 2.1. The behavior of a facet can also be represented as a system, $Beh(X) \xrightarrow{\beta} A \times Beh(X)$ (again cf. Section 2.1). A domain is a special facet whose behaviors are similarly represented as a system (Z, ζ).[14] However, the system of behaviors represented by a domain is final because for all systems represented by facets extending that domain, there is a unique morphism $(Beh(X), \beta) \to (Z, \zeta)$ defined by "is a subset of". The *discrete* domain is also considered to represent the system of all behaviors of all discrete systems. Following the definition in Section 2.1, the system of all behaviors is a final system, thus the *discrete* domain is final by definition as well.

Having shown that facets and domains can be represented by system functions, the next step is to demonstrate that these functions are in fact functors. Each facet represents a system (X, ξ) and is a function $X \xrightarrow{\xi} \Sigma X$. Table 2 provides some system signatures that are also functors. The signature of the system of behaviors of a facet is indeed a functor, T, over the sequence of observations, as $\Sigma = O \times -$ for any facet where O consists of every observable variables (can be input, output or state variables). The T-coalgebra is given by (X, fnc) where X consists of the observable variables (or state descriptors), and $fnc :: X \to T(X)$ is the coalgebraic structure.

We claim that the final coalgebra of the *discrete* domain has the following coalgebra structure:[15]

$fnc = \langle curStateD, possNextD \rangle :: States^N \to States \times States^N$

where $States^{\mathbb{N}}$ *indicates the set of discrete sequences of values from* $States$.

As stated previously and as in Example 1, the elements of the final coalgebra give the observable behaviors of all discrete systems. For any discrete process[16], (X, ξ, x), the behavior is an infinite list of observable variables, i.e. $(x_i)_{i \in \mathbb{N}}$ where x_i represents all observable variables. It is important to mention here that $s (\in States)$ is one of the observable variables of x_i. Also note that we overload our notation to have X both represent a state descriptor[17] or val-

[14] In the case of the *discrete* domain, both system representations (direct or behavioral) are possible because *discrete* is state-based. However, when a domain is not state-based, direct representation with a system signature may not be possible. For this reason, for domains, we automatically use the behavioral system without trying to represent the domain as a system directly.

[15] This coalgebraic structure is the abstracted one, i.e. where only the variables that must be present in all observations are shown.

[16] See definition of process in Section 2.1.1.

[17] A state descriptor does not give specific values, but conditions that hold for a set of values.

ues of observed variables. The set of all behaviors of all discrete processes with respect to domain variables, i.e. s, is therefore $States^{\mathbb{N}} = \{f : \mathbb{N} \to States\} = \{(s_i)_{i \in \mathbb{N}}, s_i \in States\}$. The system of all discrete behaviors is given by $\zeta : States^{\mathbb{N}} \to States \times States^{\mathbb{N}}$. In Rosetta, since in the *discrete* domain, $s \in States$ must always be observed, the system is therefore defined as $\zeta : States^{\mathbb{N}} \to States \times States^{\mathbb{N}}$ with $\zeta = fnc = \langle curStateD, possNextD \rangle$ as defined above. In other words, the *discrete* domain minimally defines a function that gives the current value of the observable variable s of a discrete system as well as the sequence of following values of s. This sequence represents the next state of the system of all behaviors. $curStateD$ provides the descriptor for what can currently be observed (the value of s from the extending facet instantiated with currently known values), while $possNextD$ describes the remaining sequence of observations. It is interesting to note that a state descriptor can be used to describe the list of future observations, as the latter is the next state of the system of all behaviors described by the *discrete* domain.

Consider the following model of a flip-flop. It has three parameters, *choice*, *input* and *output*. As their names indicate, *input* and *output* represent the input to the flip-flop and the output of the flip-flop respectively. *Choice* provides the control that decides whether the output is the current input or the value of the previous state. $Internal$ stores the value of the input when *choice* was last true.

```
facet flipflop(choice::in bit;input::in bit;
               output::out bit) :: discrete is
  internal::bit;
begin
  t1: internal' = if choice then input
                              else internal
                  end if;
  t2: output' = internal';
end facet flipflop;
```

The system defined by the $flipflop$ facet is the function

$$S \xrightarrow{\xi} \langle States, bit, bit, bit, bit \rangle \times S,$$

where S represents the set of states. In other words, the $flipflop$ facet defines a stream system, i.e. a system that on taking a transition yields observations of an element of type $States$ (s), four elements of type *bit* (*choice*, *input*, *output* and *internal*) and a next state. We distinguish between the set of abstract states of the system S and the set of states $States$ used in the specification of the system. Abstractly, we can think of using the elements of $States$ to identify some abstract states. Note also that although we use the modes “in” and “out” with parameters, all parameters are observers of the state. “In” and “out” simply add a condition to the parameters they decorate. In this example, nothing proscribes from including input and output parameters as observers of state.

The behavior of a specific process of the $flipflop$, for example (S, ξ, s_0) is given by:

$(\langle s_0, choice_0, input_0, output_0, internal_0\rangle,$
$\langle s_1, choice_1, input_1, output_1, internal_1\rangle,$
$\langle s_2, choice_2, input_2, output_2, internal_2\rangle, \ldots)$

Therefore the behavior of the $flipflop$ system is the following system:

$(\langle States, bit, bit, bit, bit\rangle)^{\mathbb{N}} \xrightarrow{\zeta}$
$\langle States, bit, bit, bit, bit\rangle \times (\langle States, bit, bit, bit, bit\rangle)^{\mathbb{N}}$

The coalgebraic structure for the behavior of the $flipflop$ model is given as:

$(States \times bit \times bit \times bit \times bit)^{\mathbb{N}} \rightarrow$
$(States \times bit \times bit \times bit \times bit) \times (States \times bit \times bit \times bit \times bit)^{\mathbb{N}}$

We use the diagram in Figure 5 to define the extension that is expressed in the $flipflop$ model as well as the morphism from the system to its behaviors. The function f represents the abstraction from an observable behavior in the $flipflop$ facet to the observation of *discrete* domain variables only. The function *map* applies function f recursively to each member of the sequence $(States \times bit \times bit \times bit \times bit)^{\mathbb{N}}$ to obtain the sequence $States^{\mathbb{N}}$. Note that the $States$ type is found both on the left and on the right of the diagram. This indicates that the same observations of the domain variable are made at the facet level as at the domain level. The function obs_fnc describes the first observation of a behavior as well as one single next step that gives us the rest of the observations of the behavior. It is indeed a pair of functions:

$curStateF :: (States \times bit \times bit \times bit \times bit)^{\mathbb{N}} \rightarrow$
$States \times bit \times bit \times bit \times bit$
$possNextF : (States \times bit \times bit \times bit \times bit)^{\mathbb{N}} \rightarrow$
$(States \times bit \times bit \times bit \times bit)^{\mathbb{N}}$

$$\begin{tikzcd}
X \arrow[r, "\text{Beh}"] \arrow[d, "\xi"'] & (\text{States} \times \text{bit} \times \text{bit} \times \text{bit} \times \text{bit})^{\mathbb{N}} \arrow[r, "\text{map(f)}"] \arrow[d, "\text{obs_fnc}"'] & \text{States}^{\mathbb{N}} \arrow[d, "\langle \text{curStateD, possNextD} \rangle"'] \\
\langle \text{States,bit,bit,bit,bit} \rangle \times X \arrow[r, "\text{id} \times", "\text{Beh}"'] & (\text{States} \times \text{bit} \times \text{bit} \times \text{bit} \times \text{bit}) \times (\text{States} \times \text{bit} \times \text{bit} \times \text{bit} \times \text{bit})^{\mathbb{N}} \arrow[r, "\text{f} \times \text{map(f)}"] & \text{States} \times \text{States}^{\mathbb{N}}
\end{tikzcd}$$

Figure 5: Commuting diagram: facet system to system of behaviors to domain

The properties of both functions composing obs_fnc are derived from the $flipflop$ facet. Using the α-denotation for terms (Section 3.2), the denotation

$t1$	$T[\![internal' = if\ choice\ then\ input\ else\ internal\ endif]\!]\ envt \equiv$ $if\ \Big((envt\_value(internal))\ \big((envt\_value(next))\ (envt\_value(\alpha))\big)$ $= \Big(if\ (envt\_value(choice))\ (envt\_value(\alpha))$ $then\ (envt\_value(input))\ (envt\_value(\alpha))$ $else\ (envt\_value(internal))\ (envt\_value(\alpha))\Big)\Big)$ $then\ True\ else\ False$
Simplified Result	$internal(next(\alpha)) = if\ choice(\alpha)\ then\ input(\alpha)\ else\ internal(\alpha)$
$t2$	$T[\![output' = internal']\!]\ st \equiv$ $if\ \Big((envt\_value(output))\ \big((envt\_value(next))\ (envt\_value(\alpha))\big)$ $= (envt\_value(internal))\Big((envt\_value(next))(envt\_value(\alpha))\Big)$ $then\ True\ else\ False$
Simplified Result	$output(next(\alpha)) = internal(next(\alpha))$

Table 4: Denotation of terms from the $flipflop$ facet

of terms $t1$ and $t2$ are given[18] in Table 4. The "Simplified Result"[19] gives the final value of the denotation. For a facet to be consistent, all of its terms must be true. This implies that the if-function needs to evaluate to $True$. This case only happens when the condition of the if-function is true. Thus, the simplified result, i.e. what the term denotes, is the condition of the if-function.

Although we now know exactly what a term denotes, we still cannot give a direct definition of the function obs_fnc. We can however use the result of the denotation to define properties of obs_fnc. Applying $currStateF$ to a sequence of states may have the following values:
Assuming Seq is a sequence of state descriptors or observable variables,

$currStateF(Seq) = s_0$ where s_0 indicates an initial state or a state independent of previous states, i.e. all variables of previous states have known values.

$currStateF(Seq) = s_n$ where $n \in \mathbb{N}$ and s_n carries information from the previous state.

[18] $(envt\_value(next))\ (envt\_value(\alpha))$ applies the environment value of the $next$ relation to the environment value of α. $(envt\_value(variable\_name))\ \big((envt\_value(next))\ (envt\_value(\alpha))\big)$ gets the function, corresponding to $variable\_name$, that maps a state to a value, from the environment, and applies it to the result of applying $next$ to α.

[19] The following shortcut is used: $variable\_name$ represents $envt\_value(variable\_name)$.

Each sequence ($\in Seq$) represents a behavior of the flip-flop system. Although the exact values of the variables in the $flipflop$ facet may not be known, with the help of the terms, it is still possible to represent a behavior of the facet. We therefore talk about state descriptors and about instantiated facets. A facet is completely instantiated when all of its variables[20] have known values. A facet is partially instantiated when the values of only some of its variables are known. Going back to the coalgebra structure, $(States \times bit \times bit \times bit \times bit)^{\mathbb{N}}$ represents the set of all sequences of the 5-tuple. The function obs_fnc then gives an observation of the facet and the rest of the behavior from that observation point. It can be said that the function obs_fnc selects only the sequences that represent behaviors of the flip-flop system.

Assuming we know the following sequence of observations for the "input" parameters $\langle choice, input\rangle$:

$\langle 1,1\rangle, \langle 0,0\rangle, \langle 1,0\rangle, \langle 0,1\rangle, \langle 1,1\rangle, \langle 1,0\rangle$.

Then $currStateF$ is defined as:

$currStateF(Seq) = s_0$ where the following properties of s_0 hold:

$s(s_0) = s_0$, Note that in future s may have a specific value - for example 0
$choice(s_0) = 1$,
$input(s_0) = 1$,
$internal(s_0) = X$,
And $output(s_0) = X$, where X represents an unknown value.

Or,

$currStateF(Seq) = s_3$ where the following properties of s_3 hold:[21]

$s(s_3) = s_3$,
$choice(s_3) = 0$,
$input(s_0) = 1$,
$internal(s_3) = 0$,
And $output(s_3) = 0$.

With the same sequence as above, when $currStateD = s_0$,

$possNextD =$

$\langle choice(s_1), input(s_1), 1, 1\rangle$,
$\langle choice(s_2), input(s_2), output(s_2), internal(s_2)\rangle$
where $output(s_2) = internal(s_2)$
and $internal(s_2) = if\ choice(s_1)\ then\ input(s_1) else\ 1$,
$\langle choice(s_3), input(s_3), output(s_3), internal(s_3)\rangle$
where $output(s_3) = internal(s_3)$
and $internal(s_3) = if\ choice(s_2)\ then\ input(s_2)\ else\ internal(s_2)$,
$= if\ choice(s_2)\ then\ input(s_2)\ else$

[20] We consider the parameters to a facet to be variables of the facet.

[21] As s_3 is in the sequence representing the behavior of the system starting at s_0, then there exists a sequence representing the behavior of the system starting at s_3. Thus $currStateF(Seq)$ can give s_3 as result.

$$if\ choice(s_1)\ then\ input(s_1)else\ 1,$$

...

6 Coalgebras of interaction

A Rosetta interaction is a special construct that allows specifying domain interaction with the help of special functions. It takes two facets and produces a third facet, typically in the domain of one of the original facets. The conceptual idea is that starting with facets $F :: T$ and $G :: R$, a Rosetta interaction could be used to generate $F' :: T$ and $G' :: R$ such that both F' and G' contain the original F and G respectively as well as information resulting from any interaction between F and G. In Figure 6, a visual representation of an interaction is given on the left. On the right, an equivalent approach to getting F' and G' is shown. Apply the R to T function on G to obtain the T image of G (if it exists) and then compose this image with F to get F'. G' can be obtained similarly.

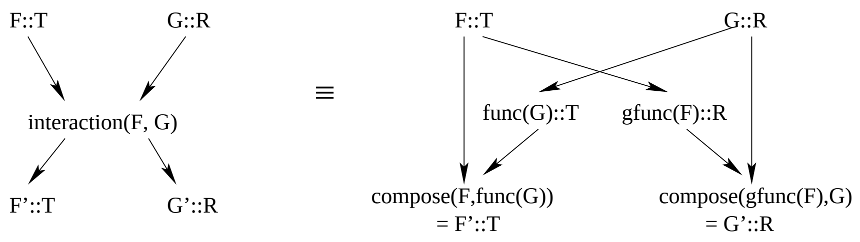

Figure 6: Interaction

To achieve a Rosetta interaction, we define functions on sets and relations between sets. We use such functions to map a facet from one domain to a facet in some other domain. Assuming objects A_1, A_2 are in set C_1 with map $f : A_1 \mapsto A_2$, objects B_1, B_2 are in set C_2 with map $g : B_1 \mapsto B_2$, then function F mapping A_1 to B_1 and A_2 to B_2 can also be applied to function f to get g. If such a function is present between the sets associated with two domains, then a commuting diagram can be formed between the coalgebraic structures of each domain. Figure 7 gives an example of defining a function from a CSP coalgebra structure to a discrete one.

Assuming there exists a function between two domains, then there exists a map between the coalgebra of any facet from the one domain to a coalgebra of a facet in the other domain. The two commuting coalgebraic diagrams of the extensions of the two domains by the two facets can thus be composed to form a cube (Figure 8). The domain function provides the "glue" between the two commuting diagrams.

$$\begin{tikzcd}
\overset{\text{CSP}}{(E \times \{E\})^N} \arrow[r, "F'_R"] \arrow[d, "\langle\langle \text{ctrlObsC} \times \text{curEvtsC} \rangle, \text{futEvtsC}\rangle"'] & \overset{\text{Discrete}}{X^N} \arrow[d, "\langle \text{curStateD}, \text{possNextD} \rangle"] \\
E \times \{E\} \times (E \times \{E\})^N \arrow[r, "F_R"] & X \times X^N
\end{tikzcd}$$

Figure 7: Functions between csp and discrete domains

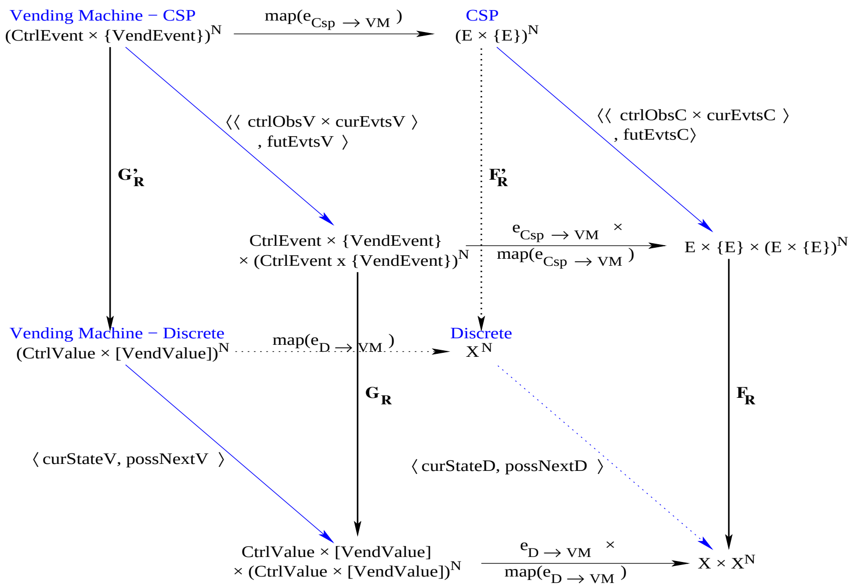

Figure 8: Cube: composition of two commuting diagrams

The top square represents the commuting diagram mapping the coalgebra of a *csp* vending machine facet to that of the *csp* domain. The bottom square also represents a commutative diagram, but for the relation between a discrete vending machine facet and the *discrete* domain. The vertical arrows represent special functions that map elements across domains. As the cube commutes, it is possible to express the function G_R (over facet) in terms of the function F_R (over domains).

7 Conclusion

Rosetta is a systems level design language that uses facets as units of specification. To formalize the semantics of facets, we define a relationship between facets and coalgebras, with the help of denotational functions. We call this definition α-denotation and it is composed of two parts. The first part consists of mapping the behaviors of a facet to a coalgebra structure of the form $X \xrightarrow{\xi} O \times X$ where X is the sequence of observations of the facet (representing behaviors of the facet) and O represents some specific values of its observable variables. The second part of the denotation defines valuation functions that map a syntactic construct in Rosetta into the corresponding value in the λ-language understood by the coalgebra. The valuation functions are used on the terms of a facet to give functions in the λ-language that define properties of the function ξ. Given the coalgebra of a domain D is final, the "extension" relations between that D and facets extending it, or between D and domains extending it are defined coinductively with the help of commuting diagrams. Commuting diagrams are also used to define functions between facets, given functions between the domains the facets extend exist.

In the past, algebras were mostly used for representing the semantics of specifications. Ehrig and Mahr [Ehrig and Mahr 1985] define the relationship between algebras and specifications. Van Horebeek and Lewi [Van Horebeek 1989] define algebras for abstract data types and talk about term algebras as well as word algebras. Although the formal semantics of most specification languages is defined with algebras, for Rosetta, we use coalgebras, which are the duals of algebras. Coalgebras are well suited for representing state-based systems. Although Rosetta is not completely state-based, the use of coalgebras does help in the formalization of the language. Furthermore, coalgebras work even for non-state-based Rosetta when we focus on the behaviors of systems instead of the systems directly. The use of coalgebras as semantics is not new to our work. The main semantic properties of final coalgebras were introduced from Aczel's [Aczel 1988] work on "non-well-founded set". Turi [Turi 1996] uses coalgebras in defining semantics. More specifically, he defines a functorial operational semantics for a syntax T and a behavior B by using a monad that "lifts" the syntactical monad T to the coalgebras of the behavior endofunctor B.

In this paper, we briefly describe how commuting diagrams can be used to define functions between facets and domains. These functions are needed in the definition of interactions between facets. Different types of interactions can exist between two system facets and some may be quite complex. Our work on defining some of these interactions with coalgebras is ongoing.

Acknowledgements

This research is sponsored in part by NSF award CCR-0209193 and The University of Kansas Information and Telecommunication Technology Center.

References

[Aczel 1988] Aczel, P.: "Non-Well-Founded Sets"; Center for the Study of Language and Information, 14 (1998)

[Alexander et al. 1999] Alexander, P., Barton, D., Kong, C.: "Rosetta Usage Guide"; The University of Kansas, Kansas, USA (1999) `http://www.sldl.org`.

[Alexander and Kong 2001] Alexander, P.,Kong, C.: "Rosetta: Semantic Support for Model-Centered Systems-Level Design"; IEEE Computer, 34, 11 (Nov 2001), 64-70.

[Allison 1986] Allison, L.; "A Practical Introduction to Denotational Semantics"; Cambridge University Press (1986).

[Burch et al. 2001] Burch, J.R., Passerone, R., Sangiovanni-Vincentelli, A.L.: "Overcoming Heterophobia: Modeling Concurrency in Heterogeneous Systems"; Proc. 2nd International Conference on Application of Concurrency to System Design (June 2001).

[Ehrig and Mahr 1985] Ehrig, H., Mahr, B.: "Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics"; Springer-Verlag / Berlin (1985).

[Jacobs and Rutten 1997] Jacobs, B., Rutten, J.: "A Tutorial on (Co)Algebras and (Co)Induction"; EATCS Bulletin 62 (1997), 222-259.

[Kurz 2001] Kurz, A.: "Coalgebras and Modal Logic"; Lecture notes ESSLLI'01, `http://www.cwi.nl/~kurz` (Oct 2001).

[Lee 1998] Lee, E.A., Sangiovanni-Vincentelli, A.: "A Framework for Comparing Models of Computation"; IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, 17, 12 (Dec 1998), 1217-1229.

[Stoy 1977] Stoy, J.E.: "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory"; The MIT Press (1977).

[Turi 1996] Turi, D.: "Functorial Operational Semantics and its Denotational Dual"; PhD. thesis, Free University, Amsterdam (June 1996).

[Van Horebeek 1989] Van Horebeek, I., Lewi, J.: "Algebraic Specifications in Software Engineering: An Introduction"; Springer-Verlag / Berlin (1989).

A Duality: state functions or variable functions

In Allison's denotational semantics [Allison 1986], a state is a function of variables to values, with the set of states being a set of functions.

$$a\_state = Var \rightarrow Values \text{ and } States = \{Var \rightarrow Values\}$$

Let $Var = \{x, y, z\}$, $Values = \{v_1, v_2, v_3\}$ and $States = \{Var \rightarrow Values\} = \{s_0, s_1, s_2\}$. Assume the following mappings:

$s_0(x) = v_1$	$s_0(y) = v_2$	$s_0(z) = v_3$
$s_1(x) = v_3$	$s_1(y) = v_2$	$s_1(z) = v_2$
$s_2(x) = v_2$ $s_2(x) = v_3$	$s_2(y) = v_2$	$s_2(z) = v_3$

Analysis of the above example shows that a dual set of functions to the state functions can be defined. The dual set is the set of variables Var, with each variable now being a function of $States \rightarrow Values$. Therefore, with $Var = \{States \rightarrow Values\} = \{x, y, z\}$, $Values = \{v_1, v_2, v_3\}$ and $States = \{s_0, s_1, s_2\}$, the duals to the above functions are:

$x(s_0) = v_1$	$x(s_1) = v_3$	$x(s_2) = v_2$ $x(s_2) = v_3$
$y(s_0) = v_2$	$y(s_1) = v_2$	$y(s_2) = v_2$
$z(s_0) = v_3$	$z(s_1) = v_2$	$z(s_2) = v_3$

B Abstractness of domains

A domain D is said to be more abstract than a domain F if domain D defines less constraints. Observations are inversely proportional to constraints in that the lesser the constraints, the more can be observed. Assume D is a tuple $\langle l_D,\ O_D,\ D_D,\ T_D\rangle$ and F is a tuple $\langle l_F,\ O_F,\ D,\ T_F\rangle$.[22] Since domain F extends D, all observations made in F are also made in D. However, there are observations made in D that are not made in F. If variable var is declared in F and not in D, then all observations made in F or in facets extending F contain var. D has all the observations of F, as well as other observations that do not involve var.

[22] Each domain/facet is a tuple $\langle label,\ observed\_variables,\ domain,\ terms\rangle$.