1 Introduction

The Rosetta systems level design language [1, 2] is an emerging standard designed to address the integration of information across domains. Specifically, Rosetta directly supports specifying and integrating models whose basic computation models differ due to technology domain. Rosetta achieves this by providing a framework where models of computation and domain specific information can be defined and organized into ontologies. Rosetta domains provide domain specific specification capability that allows the definition of basic semantic definition units, models of computation, and engineering design and analysis domains. Rosetta users then extend these domains to define specific system models and combine models for analysis using a specification algebra. Similar to all languages, Rosetta has a syntax and a semantics, where the syntax is given by a grammar and defines what expressions are valid, and its semantics gives meaning to the syntax. However, due to Rosetta’s capability for integrating multiple domains, its semantics is not fixed like for other languages. It varies with the domain. In this paper, we propose a formal semantics for the Rosetta specification language using a denotational approach.

In a previous paper [7], we provided an overview of the Rosetta language. We presented our application of units of semantics, described several models of computation and defined the corresponding domains in Rosetta. We used existing relations between domains, mainly extensions, to build a semi-lattice. In this paper, we define a formal semantics for the language. We use a notion of decomposition of semantics into modules. A first semantic module defines denotational functions for the framework. These functions are then adapted to address the specifics of domains corresponding to units of semantics. A different semantic module is defined for each such unit of semantics. All models of computation whose domains extend the same unit of semantics use the same denotational functions.

In the next section, we describe some of the terms we use in this paper. We then define the denotational functions that provide semantics to the Rosetta framework. The description of denotational functions for state-based and event-based units of semantics follows. We finally conclude the paper with some related work and future work.

2 Definitions

When talking about Rosetta, we use terms with which some readers may not be familiar with. To avoid any confusion, we define some of them here.

A model of computation is a distinctive paradigm for computation, communication, etc.

A semantic domain is a set of mathematical objects used to model agents.

*This work was supported in part by National Science Foundation Grant CCR-0209193
A unifying semantic domain is a semantic domain that can be used to represent a variety of different computation paradigms. Examples of unifying semantic domains include Tagged Signal Model and the operational semantics underlying the Ptolemy II simulator. [4]

A unit of semantics defines a unifying semantic domain.

A domain is said to extend another when its signature and presentation extend the other’s. [11]

3 Semantics of the Rosetta framework - prelude

In Rosetta, the basic unit of specification is a facet. A facet is a parameterized theory presentation with at its core the quotient algebra defined with respect to equational reasoning [11]. The facet is represented as an item that has a label, a domain it extends, a list of parameters, a list of declarations and a list of terms over those declarations and parameters. Terms are boolean expressions that describe constraints on declared items or expressions that compose and include other facets. A facet algebra provides semantics for composition activities, but its specifics are not necessary for this presentation. The domain that a facet extends provides the vocabulary and semantics used in the facet. Each domain in Rosetta is a special facet that defines the semantics for a domain-specific modeling activity.

Semantically, the Rosetta specification language can be divided into modules, with each module corresponding to a domain. Figure 1 shows such a decomposition. The framework provides the semantic foundation upon which units of semantics are defined. Semantic modules for domains corresponding to models of computation are then built upon those of units of semantics. In some cases, there may exist some semantic overlap between two modules (shaded area), indicating a possible interaction between domains. We start our definition of the formal semantics of Rosetta by providing denotational functions for the framework.

We use the term “denotation” in the sense of The Scott-Strachey Approach to Programming Language Theory [17]. We provide “semantic valuation functions” that map syntactic constructs in the program to the abstract values that they denote.

The framework of Rosetta consists of the base language and provides a semantics for the language. It can be used by itself in a way similar to the PVS [14] specification style. Each term defined in the framework evaluates to a boolean and must state a truth. Denotational functions applied to the terms give their values. We define three types of functions, an evaluation function for expression, $E$, an evaluation function for constants and literals, $V$, and a function for binary operators, $O$.

The expression evaluation function is defined as $E[e] : Store \rightarrow Values$ such that, given a store, it evaluates an expression to its value. $Store$ 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 the store 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 - it can be $univ$ or a function value (a lambda expression). Function $V : Constants \rightarrow Values$ is the evaluation function
for constants/literals. Valuation functions for operators are in the form $O[\Omega] : Opr \rightarrow \text{univ} \rightarrow \text{univ} \rightarrow \text{univ}$.

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

- $E[\xi]_{st} = (\text{st}_\text{value } \xi)$
- $E[v]_{st} = V[v]$  
- $E[\varepsilon \Omega]_{st} = O[\Omega](E[\varepsilon]_{st}, E[\varepsilon']_{st})$  
- $O[=] = \lambda(v_1, v_2). \text{if } v_1 = v_2 \text{ then } True \text{ else } False$

where  
- $\varepsilon$ is an expression  
- $st$ is the store  
- $\Omega$ is a binary operator  
- $\lambda(parameters_{\text{Imbd_functn}}).body_{\text{Imbd_functn}} : \text{type}_{\text{Imbd_functn}}$ is a lambda expression of type $\text{type}_{\text{Imbd_functn}}$  
- $\xi$ represents identifiers  
- $v$ represents constants  
- $(\text{st}_\text{value } x)$ is a shortcut for saying the value of $x$ in the store $st$)

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 denote the facet 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 $T : \text{Terms} \rightarrow \text{Store} \rightarrow \text{boolFnc}$, where $\text{boolFnc}$ is a function $\text{Vars} \rightarrow \text{Boolean}$ with $\text{Vars}$ 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 store. 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 store $st$ contains the following mappings:

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

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

- $T[x = 2]_{st} \equiv O[: V[2]]_{st} \equiv \text{if } (2 = 2) \text{ then } True \text{ else } False \equiv True$
- $T[(myFnc 1) = 2]_{st} \equiv O[\equiv]((\lambda x : \text{int}. \ (x + 1))2) \equiv True$
- $T[(\text{undefFnc } x) = 2]_{st} \equiv (\text{undefFnc } 2) = 2 \ - \ \text{boolean equation with one unknown}$

The framework is encoded in the logic domain and provides the mathematical and logical basis to the language. Additional domains extend logic to provide more computational capabilities. Therefore, more case by case valuation functions need to be given for each domain. This is done in the semantic module corresponding to the domain.
4 Denoting Units of Semantics

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 [4]. 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.

The vocabulary in the case of a Tagged Signal Model (TSM) consists of values, tags, events, signals and processes. Its 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). For example, the TSM where firing of processes is done sequentially according to events ordered with respect to a global clock is different from one where firing of processes follows a time warp approach, i.e. parallel discrete event simulation.

The vocabulary of a state machine consists of inputs, outputs, states and transition functions. In this case, the semantics describes states with respect to the inputs, outputs and transition functions. The semantics of a state machine is that of an automaton. A change of state occurs when a transition function is fired on the occurrence of an input (assuming no epsilon-transition). An output may also be generated. However, there may be other semantics for the same vocabulary of inputs, outputs, states and transition functions. For example in a timed automaton, the transition functions also depend on some clock constraints.

4.1 State based model

We identify the vocabulary of a state based model to consist of a set of states (finite or infinite, denumerable or nondenumerable), a finite set of inputs, a finite set of outputs and a transition relation that given a state returns a next state or a set of next states. 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.

\[ K \text{ Set of states} \]
\[ \Sigma \text{ Set of alphabets (inputs and outputs)} \]
\[ \delta \text{ Transition function: } K \times \Sigma \rightarrow K \] – it is understood that \( \Sigma \) represents the input alphabet here

The first step in defining the semantics of our state based model is to define what a state denotes. Following the definition by Allison [3] for sequential execution of a program, a state defines the current values of variable identifiers in a specification or program.

\[ S = \{ \text{Var} \rightarrow \text{Values} \} \]

In other words, the set of states 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 [3] defines a command to denote a relation \( (S \rightarrow S) \). He then gives a valuation function for commands \( (C : Cmd \rightarrow (S \rightarrow S)) \). He also describes the notion of expressions along with a function that evaluates the expression value in a given state \( (E[e] : S \rightarrow \text{Value} \text{ where } [e] \text{ represents semantic brackets}) \). Several differences exist between an expression and a command. A command does not have an intrinsic value as does an expression. A command also causes a state transformation whereas evaluating an expression does not always (as long as expression does not have side effects). However, when expressions have side-effects, the valuation function is modified to \( E : Exp \rightarrow Value \times S \) to give \( E[e] : S \rightarrow Value \times S \).

Although our state based model is not restricted to defining sequential program execution, we can mirror its semantics on Allison’s definitions. However, there is a major difference between our definitions and Allison’s. There is no command in a declarative language because there is no notion of assignment or side-effects. The side-effect of a command is explicitly defined in a declarative language 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 variables.

The vocabulary and semantics associated with the state-based unit of semantics are that of a state machine. The vocabulary consists of:
As discussed previously, in Rosetta, all expressions are purely boolean. There is no side-effects and no relation. In event-based there is no specific construct, however, we do associate semantics with events in the state-based model, we chose to establish semantic definition on a specific construct - the state transition semantics of an event-based model can be associated with the processing of events in signals. In the case of events and a set of signals. Signals can be grouped in sequences to represent a behavior of a process. The define concurrent models of computation.

Our event based model is based on the Tagged Signal model [8]. The Tagged Signal model provides an approach where the denotation of a process is a partial function or relation on signals. A signal is a set of all input and output signals at a specific instant. Our event-based domain provides the vocabulary to define concurrent models of computation. Following Lee and Sangiovanni-Vincentelli’s Tagged Signal Model [8], an event is a member of the set $T \times V$ with $T$ being a set of tags and $V$ being a set of values. A signal $s$ is defined as a set of events, therefore, a subset of $T \times V$. A process contains a number of input and output signals and is represented as a set of tuple of signals. Each tuple represents a behavior of the process as it keeps track of the state of all input and output signals at a specific instant. Our event-based domain provides the vocabulary to define each behavior of the process individually. Indeed, we model the process from behavior to behavior. Therefore, the definition of a process is implicit and we do not need a process object in our vocabulary. Furthermore, the state of a process is dependent on events only, i.e. the process does not have state variables other than what is needed for the process to know what events are expected. Thus as a state marker is used
to mark the state a process is in, the only variable in the state is that marker. We use some of the evaluation
functions previously defined in Section 3 and add some new ones.

The vocabulary associated with the event-based unit of semantics is mainly that of concurrent systems. It consists of:

- Tags Set of tags.
- Values Set of values.
- Events The set of events.
- States The set of states. Each state is given by an event.
- Signals The set of signals.
- @ Similar to the apply function and is used for dereferencing signals in states indexed by events.

The semantics associated with that vocabulary consists of:

- Events : \{Tags × Values\} Set of pairs of tags and values.
- previousEvt : Events = Tags × Values A particular event is a particular pair of tag and value. previousEvt is the last event processed. It is used as an index to the last state of the process.
- evt Represents current state and is in fact the event being processed in that state.
- Signals : φEvents Set of sets of events.
- sig : Signals = \{Events\} A particular signal sig is a set of events.
- getEvent : Signals → Events The next event obtained from events in signals. The event returned by getEvent is the current event to be processed. It is used as a reference to the current state.
- States : \{SignalLabel → SignalValue\} States in event-based are dependent on events and are used to keep track of values of signals.
- EventBasedState : Events ↵ States Each state is uniquely referred to by an event.
- @ : SignalLabel → Events → (Events → States) → SignalValue Given a signal label and an event, the value (contents) of the signal is obtained by first finding the state corresponding to the event, and then applying the state function to the signal label to find the signal value. The @ operator therefore returns a lambda expression expecting a function (Events → States) as parameter. This function helps find the state corresponding to the event.

The previousEvt variable acts as the marker that keeps track of the state a process is in as it stores the particular event that was last processed. By looking at the event’s tag and value, we know exactly in which state the process is. Thus, given the next event, we know what the next behavior of the process should be. The next event becomes the current event and it can be obtained by different algorithms. For example, it can be the event with the lowest tag within a particular signal (obtained by applying the getEvent function on that signal), or it can be the smallest event among several signals.

The valuation functions for the event-based domain are similar to those of the state-based. However, due to the difference in semantics in states, there are some succinct variations, especially in the @ operator. St represents the store and st_value returns the value of the parameter as is from the store.

\[
O[\sigma][\sigma][\sigma] = (\lambda(\sigma1, \sigma2). (\lambda(\sigma1, \sigma2) (\sigma1, \sigma2) \sigma1)):
\]

SignalLabels → Events → (Events → States) → SignalValue

\[
E[sig@evnt] =
O[\sigma][\sigma][\sigma] sig (E[\sigma][\sigma][\sigma][\sigma] st (st_value EventBasedState)) =
O[\sigma][\sigma][\sigma] sig (st_value evnt) (st_value EventBasedState) =
(\lambda(\sigma1, \sigma2) (\sigma1, \sigma2) (\sigma1, \sigma2) \sigma1)) (st_value EventBasedState) =
((st_value EventBasedState) (st_value evnt)) sig
\]

As mentioned above, each event is uniquely associated with a state through the function EventBased–State : Exp. However, this mapping is not shown to the user. The latter is only aware that signals are dereferenced according to a specific event. It is in the denotation function that the mapping from a state to an event is done.

### 4.3 State-based vs event-based

The major difference between state-based and event-based domains is the fact that in state-based, terms are used to define properties of the next function, i.e. the state transformation relation, while in an event-based
domain, terms are used to define the processing of an event in the current state. Therefore, although both state-based and event-based domains have the notion of states, the semantics associated with a state differs from one domain to the other. In state-based, the notion of state depends on the variables of the system. Thus, the state transformation relation is a function of state variables. In other words, the definition of the next function is given by the change in value of system variables. In event-based, a state is associated with an event. There is no next function per se. The next state is given by the next event, which is usually already present in a signal of the process. Another difference is that in signal-based domains, variables other than signals cannot be dereferenced to a state that is not the current state. This distinction is a direct result of the previous one. In state-based, variables control state transformations, and it is important to be able to dereference variables across different states. Knowledge of the next state (or future states) depends only on how variables evolve. In event-based, the next state is known a priori. Events are usually generated by other processes or by the same process in some previous state. Since states do not depend on variables, the idea of dereferencing variables other than signals to different states is futile. Variables are only used within a state to, for example, create events within that state or to do some local calculations. Once there is a change of state, the values of variables in the state before the change become useless because in event-based, terms are used to model the behavior of a process within a particular state.

5 Related Work

Denotational semantics [15] traditionally provides formal foundation for specifying language semantics using the lambda calculus. However denotational semantics’ monolithic nature proves problematic for combining specifications as required by this effort. To address this problem, Moggi [12, 13] introduces the use of monads to modularize semantics, making it easier to change the semantic definition of a language. Monads are used to represent semantic building blocks representing different computational activities [18, 19, 20].

Several implementations followed for implementing modular interpreters based on the monad concept. Steele [16] defines a process of building interpreters using monads to represent underlying components. Espinosa’s Semantic Lego [5] defines a similar concept of semantic building blocks that are composed to define underlying language semantics. Hudak and Liang [10, 9] define higher-order monads called monad transformers used for lifting one monad into another. Similarly, Jones [6] defines monad lifting to compose independent monads to form a larger semantic structure. This lifting process moves a value in one semantic domain into another. Distributivity laws can sometimes be used as in the case of the Exception monad. This allows one particular monad to distribute over all others, allowing hierarchical composition [6]. Conceptually, this work demonstrates the effectiveness of the monad construct for defining and combining semantic properties. Researchers are able to capture independent semantic features such as state, continuations, nondeterminism, and two different calling semantics and provide mechanisms for composing semantics.

Although the definition of individual domains in this effort uses a denotational style, the composition mechanism is similar to that proposed by Hudak and Liang [10, 9] and Wadler [18, 19, 20]. Current work defines compositional mechanisms based on specification coproducts and functors that allow domain definitions to remain encapsulated.

6 Conclusion

In this paper, we present a formal semantics for the Rosetta specification language. We define the semantics by decomposing the language into modules, one for the base language and one for each domain. We use a denotational approach to provide meaning to the vocabulary of each module. Denotational functions are defined for interpreting expression, constants and literals, and binary functions. The functions for the framework are first defined and functions for domain semantic modules are built upon them. We provide functions for a state-based model and an event-based model.

Although the state-based semantics and the event-based semantics are different, it is still possible to integrate models from those two different domains. The state-based domain provides vocabulary and semantics for defining system states. The event-based domain provides vocabulary and semantics for defining concurrent system through event processing. The co-product of state-based with event-based domains gives
rise to a new domain where both events as well as the notion of state transformation of a process can be used. In this new domain, a process is allowed to have local variables that when modified, cause a state transformation of the process. These local variables need not depend on events. By projecting from the co-product into state-based and event-based, we get two systems, one focused on how variables affect state-change and the other focused on how events are processed.

The semantic module corresponding to the co-product will contain all the evaluation functions of both domains. The @ function will be overloaded with its definition from the state-based domain and its definition from the event-based domain. The complete definition of this function is still under development. Once the semantics of the co-product is defined, systems involving both state machines and concurrency can be modeled in Rosetta.

We are also working on defining the semantic modules for specific models of computation. So far, we have observed that these models of computation do not need any other evaluation functions than those of the units of semantics they extend.

References


