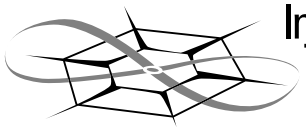
The University of Kansas



Information and Telecommunication Technology Center

Technical Report

The Rosetta Meta-Model Framework

Cindy Kong and Perry Alexander

ITTC-FY2003-30150-01

November 2002

This research was funded in part by the National Science Foundation under award CCR-0209193

The Rosetta Meta-Model Framework

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

Abstract

Heterogeneous systems are naturally complex and their design is a tedious process. The modeling of components that constitute such a system mandates the use of different techniques. This gives rise to the problem of methodology integration that is needed to provide a consistent design. In this paper, we propose a meta-model framework that provides such an integration. The semantics of different computational models can be expressed and used together in the Rosetta framework. We use denotational semantics to define unifying semantic domains, which are themselves extended to provide ontologies for models of computation. Interaction relations defined between models are then used to exploit and analyze model integration. We demonstrate our approach by providing applications where different computational models are used together.

1 Introduction

The design of computer systems increasingly involves integrating models with heterogeneous semantics. Today's systems involve components implemented using technologies such as software, digital hardware, analog hardware, optics, and MEMS (Micro Electro Mechanical Systems), each with their own domain specific semantics. When integrating such components in analysis and synthesis activities, it is necessary to bring together such disparate models to effectively predict performance properties. Thus, ontologies supporting the specification and integration of computation models must be developed and supported in next-generation systems level design languages.

The Rosetta specification language [1, 2] is being developed with such a goal in mind. It is geared towards system level design and provides a framework where ontologies of computational models can be defined. It also provides an interaction mechanism that defines inter-domain impact.

In this paper, we describe the Rosetta domain framework and use it to define models of computation. We use the notion of a unifying semantic domain [8] to provide a domain of discourse for models. A semantic domain is called unifying when it is used to represent various design paradigms. We exploit relations between unifying semantic domains and models to derive interaction models. As inter-domain interactions depend on the nature of models involved, their derivation and definitions cannot be automated. It is the domain designers' responsibility to define such interaction models. However, an interaction model need be defined only once. We present two unifying semantic domains, define a lattice of models of computation based on them, and express interactions between some of these models. We demonstrate modeling capabilities in the design of a vending machine in CSP [16]. We also show composition and interaction in modeling power consumed in a state-based timer model.

2 The Rosetta Specification Language

System level modeling involves integrating domain-specific models into a consistent system model. This integration is not easy due to the possibility of inter-domain interactions, where information from one domain-specific model impacts models from other domains. Rosetta attempts to address this problem directly in its domain representation semantics. Domains define ontologies of computational models and design paradigms. Systems are modeled by aspects, each as-

pect representing a domain specific viewpoint. A system model is then obtained by assembling these aspects with the use of interaction models. An interaction model specifies what information exchange occurs between two domains.

In Rosetta, an aspect is called a facet. It represents the basic unit of modeling. Its specification consists of a `signature`, a `domain`, and `terms`. The signature consists of the name of the facet and a list of parameters. The domain that the facet extends provides a domain of discourse for the facet. In other words, the facet uses items defined in the domain and adds additional properties to them. Terms are used to express such properties. A facet can also create new items and again, terms can be used to define their properties. Semantic correctness is then defined as consistency of terms with respect to the specified domain. If no inconsistency is introduced by terms and declarations within the facet, then the facet definition is consistent.

Syntactically, as shown below, a facet has a label, a list of parameters, a list of declarations and a list of terms. Declarations create constants, variables, functions, types and even other facets. Terms are boolean expressions that describe constraints on declared items, or define and include other facets to support structural definition. The `::` operator indicates membership and is used in declarations to provide type information. In the case of a facet declaration, the domain the facet extends provides the type information of the facet.

```
facet <facet-label> (<parameter-list>)
 ::<domain-label> is
   <declarations>
begin
   <terms>
end facet <facet-label>;
```

Domains represent computation models and vocabularies for domain specifications. When writing a specification, a designer chooses a domain appropriate for the model being constructed. The domain is extended by adding declarations and terms that use the base domain's predefined computational model. Alternatively, designers can define their own domain by extending an existing domain or start completely from scratch. The advantage of using an existing domain is reuse of the domain and its interactions with other domains. The syntax for domains is similar to that of facets as domains are simply special facets.

3 Domain Semantics

Although the Rosetta framework provides some semantics in the form of facet algebra, it is mainly a syntactic platform. Semantics used in facet specifications is provided in domains. Therefore, specifications have different meanings depending on their domains. We use a denotational approach [3, 24] to describe these domain semantics. The denotational approach provides semantic valuation functions to map syntactic constructs to abstract values. Using different valuation functions thus gives different meanings to the same construct.

There are several ways for defining denotational semantics of a program. A common approach is to provide partial functions or relations on program states [3, 24]. Each state defines values for the variables in the program. A command in the program then denotes a change in state. Lee and Sangiovanni-Vincentelli [20] propose another approach to denotational semantics based on signals. A program, or rather, a process is represented by a partial function or relation over signals. This approach to denotational semantics is specifically suited to modeling concurrency.

Program denotations into different representations give rise to the idea of making these representations available at a higher level of abstraction. By making these representations available, we allow designers to decide earlier on the best representations for their system specifications. We consider state based and signal based semantics to represent unifying semantic domains [8] that we call `units of semantics`. A unifying semantic domain represents a set of semantic objects that is used to represent several models of computation. For example, the state based semantics is used to express computational models such as finite state machines, sequential machines, continuous computation models and so on. The signal based semantics is used to express models such as CSP, Petri nets, and other concurrent models.

We use `domains` to contain the information relevant to units of semantics, models of computation, design paradigms, or engineering concepts. A domain provides an environment where new semantic definitions can be defined for new or existing constructs. Additional constraints can also be defined on existing semantics. Existing constructs or semantics need not be originally declared in the same domain. A domain is said to *extend* another one when it uses constructs and semantics defined in that other domain. This notion of extension is similar to that in the Java Programming Language [12]. The extending domain can use definitions from the domain being extended. However, the extension is not always conservative. By adding new semantics or constraining previously defined ones, it is possible to add properties that are inconsistent with existing ones. Consistency is guaranteed only if the

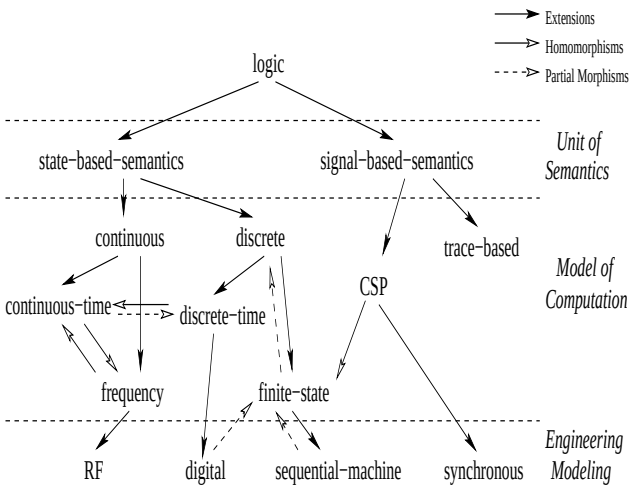


Figure 1. Lattice of Domains

properties added concern new syntactic constructs and are consistent in themselves.

Figure 1 shows the lattice of domains that is obtained through extension. The root of the lattice is the domain `logic` that acts as prelude to the Rosetta language. It contains definitions for all basic types and constructs of Rosetta. The rest of the domains are divided into three groups: *unit of semantics*, *model of computation* and *engineering modeling*. `State-based-semantics` and `signal-based-semantics` represent units of semantics that define state-oriented and signal-oriented unifying semantic domains respectively. The `discrete`, `finite-state`, `discrete-time`, `continuous`, `continuous-time` and `frequency` computational models extend state-based-semantics. `CSP` [16] and `trace-based` are two models of computation based on signal-based-semantics. Some examples of engineering modeling domains are `RF`, `digital`, `sequential machines` and `synchronous`.

Discrete domain	1

```
domain discrete(f::facet) :: state_based_semantics is
begin
  d1: exists (fnc::<*(st::States)::natural*> |
        forall(s1,s2::States|
          (s1 /= s2) implies (fnc(s1) /= fnc(s2))))
end domain discrete;
```

The `discrete` domain (Specification 1) represents models where each state is discrete. It extends `state-based-semantics` and adds an additional constraint on the `States` set (declared in `state-based-semantics`). Since each state is discrete, there exists an injective relation (or one-to-one relation) between the set of states and the set of natural numbers. A one-to-one relation involves a function `fnc` that maps a state to a natural number such that if two states are different, then they map to two different numbers. Term `d1` expresses this relation. There exists a function `fnc`, such that for any two states `s1` and `s2`, if `s1` and `s2` are different, then their mappings under `fnc` are different.

An example of a domain that extends the `signal-based` unit of semantics is the `csp` domain. It uses items introduced in that parent domain such as:

Values type representing set of values associated with tags.

Tags type representing the set of tags.

Event constructed type that consists of a value and a tag. A specific instance of the type is given as `event(t1, e1)`. Accessor functions are automatically generated to access each field of the constructed type. Therefore, `tag(event(t1,v1))` returns `t1` and `value(event(t1,v1))` returns `v1`.

Signals type representing the set of signals. Each signal is a set of events.

@ dereferencing operator of a label with respect to a tag. This operator appears in all domains, but its semantics differs.

In the `csp` domain, we add constraints over these items to express the properties that are necessary to define processes. Specification 2 provides part of the Rosetta `csp` domain. Term `c1` states that there is a total ordering of all tags in `csp` (the partial ordering axioms are also defined but not shown here). Therefore, although `Signals` are sets, due to the ordering of all tags, and therefore of all events, `Signals` can be considered as ordered sets. The function `put` adds an event to a signal. `Get` is its counterpart as it takes an event from the signal (we intend `get` to return the event in the signal with the lowest tag as shown in term `c2`. If `evt` is the event returned by *get*, then its tag is less than or equal to the tags of all other events in the same signal.) Function `getSignal` returns the signal without the event returned by `get`. Term `c3` states that two different events within the same signal cannot have the same tag value. This prevents nondeterminism as events are processed according to the lower tag value. We need not say that nondeterminism can be useful in modeling and therefore we may want to allow it in Rosetta specifications. However, for the purpose of this paper, we do not need this additional complication. Although not shown here, we also have definitions of the

different protocols for communication over signals, e.g. rendez-vous, letterbox and so on.

CSP domain | 2

```
domain csp(f::facet) :: signal_based_semantics is
  put(sig::Signals;evt::Event)::Signals;
  get(sig::Signals)::Event;
  getSignal(sig::Signals)::Signals;
  tg::Tags; // current tag
  nextTg::Tags; // next tag
  ...
begin
  c1: forall(t1,t2::Tags | t1 <= t2 or t2 <= t1);
  c2: forall(sig::Signals;evt::Event |
        (get(sig) = evt) implies
        forall(otherEvt::Event |
           (otherEvt in sig) implies
           (tag(evt) <= tag(otherEvt))));
  c3: forall(sig::Signals;evt1,evt2::Event |
        (evt1 in sig) and (evt2 in sig)
        and (evt1 /= evt2) implies
        (tag(evt1) /= tag(evt2)))
  ...
end domain csp;
```

4 Examples

4.1 Example of A Coffee/Tea Vending Machine

We choose a vending machine as a modeling example, following the tradition started by Hoare [16] and continued in several work [18, 28]. We model a vending machine that provides the choice of a cup of coffee or a chocolate when a dollar is paid and the choice of a cup of tea or a cookie when 75c is paid. The dollar and 75c events are the guards between two alternatives. Once one of these events occur, an external choice (the user) selects the next event. The CSP model is given as:
VM = (dollar → (coffee [] chocolate) → VM) ‖ (75c → (tea [] cookie) → VM)

In Specification 3, `csp_vending` is the Rosetta `CSP` specification representing the above vending machine process. `VendTags` represents the set of tags in our system. There are two possible states to the vending machine: (1) the machine is expecting money, and (2) coin event has already happened, the machine is expecting a choice. Therefore, `VendTags` consists of two elements. Since it is totally ordered, we choose integral values 1 and 2 that are intrinsically ordered. The values associated with events are `dollar`, `coffee`, `chocolate`, `75c`, `tea` and `cookie`, each representing an event from the CSP model (`VendValues`). `VendEvent` and `VendSignals` respectively define the event type and signal type used in our specification. The vending machine process uses three signals: `input` for events produced elsewhere, `output` for events produced by the process, and `ctrlSig` for control events. The control signal `ctrlSig` contains only one event at any time. The tag of the event tells the process the state in which it is and the value of the event keeps track of the last event processed. As the control signal is not accessible from outside the process, we know for sure that the current tag can take only two values, namely 1 and 2. Therefore, we can do dereferencing as follows `ctrlSig@nextTg` where `nextTg` is the next tag. The machine works as follows. Terms `c1` and `c2` define the values of `tg` and `nextTg`. Term `c3` defines the machine's behavior. If event in `ctrlSig` is `init` and if event in `input` is dollar, then `ctrlSig@nextTg` becomes a copy of the current `ctrlSig` from which event `init` was removed and event `dollarCoin` added (`input` becomes a new signal without the coin event). If instead, control event is `dollarCoin`, then depending on the event in `input`, either a coffee or a chocolate event is put in `output`. Note that `ctrlSig` and `input` are modified such that the machine goes back to the initial waiting state at the next tag.

4.2 Example of an Interaction

An interaction represents a relation that defines when and where information from one domain impacts another. Whenever facets using domains that interact are composed, the interaction specification is automatically included. We demonstrate the methodology behind interaction models with an example of high level power analysis.

Power is the leading constraint in embedded system modeling. However, as power is implementation dependent, power analysis has been mostly done at low levels of abstraction. We use Rosetta and its interaction mechanism to propose a new methodology for estimating power dissipation at high-level. The methodology consists of first modeling the behaviors of a system. Then, the interaction between the system models and technology specific power models is analyzed to derive power consumed if the system is implemented in that technology. In another paper, we provide examples of power analysis for implementation in the CMOS technology. In this paper, we are interested in analyzing power dissipated in a software implementation. Tiwari et al. [27] propose a method for estimating power consumed by processors while executing some instructions. We use their results to provide estimation models for power consumed in a Rosetta design. We analyze the operations that occur in a model. The instructions that are needed for a specific operation can usually be estimated by engineers. Using empirical models derived from Tiwari's work, we can therefore calculate the

Models for a coffee/tea vending machine | 3

```
VendTags::subtype(Tags) is {1,2};
VendValues::subtype(Values) is
 enumeration[dollar, coffee, chocolate,
             75c, tea, cookie];
VendEvent::subtype(Event(VendTags,VendEvent));
VendSignals::subtype(Signals) is set(VendEvent);
CtrlValues::subtype(Values) is
 enumeration[init,dollarCoin,75cCoin];
CtrlEvent::subtype(Event(VendTags,CtrlValues));
CtrlSignals::subtype(Signals) is set(CtrlEvent);

facet csp_vending() :: csp is
 input,output,ctrlSig::VendSignals;
 export input,output;
begin
 c1: tg = tag(get(ctrlSig));
 c2: nextTg = if (tg = 1) then 2 else 1 end if;
 c3: case value(get(ctrlSig)) is
   init ->
    case get(input) is
     event(1,dollar) ->
      (ctrlSig@nextTg =
       put(getSignal(ctrlSig),
           event(nextTg,dollarCoin))) |
    ... // case 75c event similar to dollar event
    end case |
   dollarCoin ->
    case get(input) is
     event(2,chocolate) ->
      (output@nextTg = put(output,event(2,chocolate)))
       and (ctrlSig@nextTg =
         put(getSig(ctrlSig),event(nextTg,init)))
       and (input@nextTg = {}) |
     event(2,coffee) ->
      (output@nextTg = put(output,event(2,coffee)))
       and (ctrlSig@nextTg =
         put(getSig(ctrlSig),event(nextTg,init)))
       and (input@nextTg = {})
    end case |
    ... // case 75cCoin - similar to dollarCoin case
 end case;
end facet csp_vending;
```

Interactions | 4

```
interaction DandP(f::discrete, g::power)
 :: logic is
begin
  power_to_discrete: {};
  discrete_to_power:
    reflect.meta_if
     (orModified(sel(v::meta.params(g) | output(v))),
      outputActivity=0, outputActivity=1)
    and reflect.meta_equal(operatorActivity,
      g.calculateOpActivity(meta.getOperators(g)))
    and reflect.meta_equal(structureActivity,
      reflect.meta_sum(g.calculateStructActivity
        (meta.getInstantiatedFacets(f))))
    and reflect.meta_equal(activity,
      outputActivity*g.outputActCoeff +
        operatorActivity + structureActivity);
end interaction DandP;
```

Timer model | 5

```
facet timer(set::in boolean;startTime::in natural;
 alarm::out boolean) :: discrete is
  currentTime::real;
  decrement(time::natural) :: natural is time - 1;
begin
  init:set => currentTime' = startTime;
  t1: (not set) => currentTime' = decrement(time);
  t2: alarm = if (currentTime'=0)
              then true else false end if;
```

power consumed by a model. The result is not very accurate and will also probably be underestimated. However, it should be sufficient for comparison purposes between implementation technology and between possible designs.

The `power` domain defined in Rosetta specifies how power consumption is to be calculated across a state change (`p1: power' = activity * nominal + leakage;`). Depending on the implementation technology, `activity` either represents switching activity at the gate level or current drawn by instructions. `Leakage` is a parameter provided by engineers to represent the static power dissipated through leakage current. *Nominal* is the nominal voltage. The *power′* is a shortcut notation expressing $power@next(state)$, i.e. the value of power across a state transformation. The following functions and variables are also declared in the power domain:

```
calculateOpActivity(opSet::sequence(labels))
 ::real;
calculateStructActivity(facetSet::set(facets))
 ::real;
outputActCoeff::real;
```

Specification 4 provides an interaction model of what occurs between the power and discrete domains.

A software power facet (`swp`), extending the power domain, is written. It provides terms that define the functions and variables mentioned above. It is important to note that declaration is different from definition. A declaration gives the signature of an item, while definition provides the value of a just declared or already declared item. `CalculateOpActivity` calculates the total average current (in mA) drawn by all the operations involved in a design. `CalculateStructActivity` calculates the average current drawn by instantiated facets. Facet instantiation is used in structural composition.

```
facet new_swp(Vcc,leakage::design posreal)
                              :: power is
  calOpActivity(opSet::sequence(labels)):: posReal is
    if (opSet == nil)
      then 0
      else let (operator::label be opSet(0)) in
         case operator is
```

```
          +' -> 400
          '-' -> 300
          'not' -> 276
          '=' -> 500
          '=>' -> 700
         end case
       end let
       + calOpActivity(tl(opSet))
    endif;
  calStActivity...;
begin
  t1: calculateOpActivity = calOpActivity;
  t2: calculateStructActivity = calStActivity;
  t3: outputActCoef = 0;
  t4: nominal = Vcc;
  discrete_to_power:
    if modified(alarm)
      then outputActivity = 0
      else outputActivity = 1
    end if
    and operatorActivity = calculateOpActivity(...)
    and structureActivity=calculateStructActivity(...)
    and activity = outputActivity*outputActCoef +
             operatorActivity + structureActivity;
end facet new_swp;
```

Specification 5 shows a simple example of a timer that goes off whenever it decreases to zero. Such a timer is used in operating systems for threads that are given a certain CPU time budget, but do not allow preemption. We compose this model with the software power model to analyze the power consumed by the timer as follows:

```
timer(setVal,start,alarm) DandP swp(Vcc,leakage)
```

The resulting facet is one where both models coexist. We obtain a software power augmented model (`new-swp` as shown above) by projecting this composed facet into the power domain. This results in adding terms (`discrete-to-power`) derived from interaction `DandP` into the new facet. This new facet also contains terms (`t1`, `t2`, `t3` and `t4`) and declarations (`calOpActivity` and `calStActivity`) from the original software power facet. Function `calOpActivity` denotes what `calculateOpActivity` does. `If-then-else`, `let` and `case` are functions in Rosetta. They return the value of the expression they evaluate to. For example, if the sequence `opSet` is empty (`nil`), the whole `if` expression evaluates to 0. `CalOpActivity` is a recursive function. It returns 0 if the sequence is empty, else it evaluates the current drawn by the first operator in the sequence, and adds this value to a recursive call on the rest (`tl(opSet)`) of the operators in the sequence. The values in the `case` expression are derived from Tiwari's [27] estimation of current drawn in a 486DX2 processor. For more details on the Rosetta syntax, please refer to the Usage Guide [1].

We simulate this augmented power model to estimate power consumption for the timer. `Vcc` for the 486DX2 is 5V. We assume a power leakage of 0.25 W. Since `outputActCoef` is 0, we do not care about the `outputActivity`. As no facet is instantiated in the timer example, `structureActivity` is also 0. `OperatorActivity` is equal to 3700 mA (one '-', two '⇒' and four '='). Therefore, total power is given by 18.75 Watt (3700 * 5 + 0.25). This is only an average estimate and the energy consumed can then be calculated according to the speed of the CPU. It can be made more accurate by other measurement of current drawn per instruction, and by involving only the operators that are used per state, instead of simply calculating the current drawn by all operations in the facet.

5 Related Work

Several efforts have been directed toward combining different methodologies and tools together. We group the related work into five categories: (1) computational model use and composition in one framework; (2) discrete and continuous domain modeling; (3) meta-modeling; (4) requirements engineering; and, (5) multiple logic.

Computational model composition is an important problem. Ptolemy II [7], SAL [6], and Metropolis [8] provide different approaches to this problem. Ptolemy II type system [21] is augmented by interaction types derived from analysis of automata representing concurrent computational models. Models are composed by using specific connecting entities of these types. The SAL, Symbolic Analysis Laboratory, framework allows the use of many tools and models by using an intermediate language. This language can be translated to and from the languages of different analysis tools, and can also be used to represent different concurrent computational models. The Metropolis project proposes a framework where formal models can be defined and compared. It uses trace algebra to compose models through connecting channels. Rosetta differs from these approaches in that it allows vertical integration of models. Vertical integration does not need communicating channels between models.

There are several approaches focused on combining discrete and continuous modeling. A hybrid automaton [15] is a generalization of timed automaton [4]. Continuous behavior is modeled according to differential equations within a state, while discrete behavior is modeled as jumps across states. Model-checking techniques can be applied to hybrid automata. HYTECH [26] provides a model checker for hybrid automata, while UPPAAL [5] provides a model checker for networks of timed automata. VHDL 1076.1 [9], also

known as VHDL-AMS, is a language for mixed-signal design. It is a hardware description language that supports the description and simulation of digital, analog and mixed analog/digital systems within one environment. It defines a concept of quantity, to represent continuous variation. Rosetta also provides a framework for hybrid or mixed system modeling. However, Rosetta's framework is open and allows definition and use of other computational models.

The notion of meta-modeling is an object-oriented paradigm and UML [13] is often used for its implementation. A meta-model defines objects that can be instantiated to obtain objects of models. The MultiGraph Architecture [22] is a toolkit for creating model integrated program synthesis (MIPS) that allows experts to integrate models that represent domain-specific systems through a methodology called model-integrated computing (MIC). The Generic Modeling Environment (GME) [19] provides a tool that implements the MultiGraph Architecture. Terrasse et al. propose another UML metamodeling architecture to define abstract bases of agreement for interoperability of information systems [25]. Rosetta's framework can also be considered as a meta-modeling architecture. It provides semantic meta-models for computational models.

In requirements engineering, we relate Rosetta to viewpoints, feature engineering and aspect-oriented programming. The Viewpoints [10] framework allows different perspectives of a system to be expressed with different representation. Consistency check instructions provide conditions that need to hold for composition of the viewpoints into a complete system. Feature engineering consists of describing a system (mainly in telephony) by features, with a feature being a unit of functionality [29]. The idea is to specify features as if they were independent and then use composition operators to combine them together. In Aspect-Oriented Component Requirements Engineering (AOCRE) [14], components are categorized according to the "aspects" that they either provide or need. Engineers use these aspects to reason about inter-component relationships. In aspect-oriented programming, the same notion of an aspect is applied on methods [17] instead of components. Rosetta differs from Viewpoints and feature engineering as it provides consistency checks automatically in its type checking. Rosetta does aspect-oriented modeling at a higher level of abstraction (specification instead of programming).

Institution theory and Isabelle provide approaches to combining multiple logics. The concept of an institution [11] is introduced to formalize the notion of a "logical system" and provides a foundation for exploiting the relation between logical systems. Some sentences can be consistently translated from one logical system to another. Isabelle is a generic theorem prover [23]. It provides a logical framework, i.e. a meta-logic, that can be used to formalize several objects-logic. Rosetta focuses on combining computational models instead of logics.

6 Conclusion

We introduce the Rosetta meta-model framework as a framework where different computation models can be used and composed. We provide definitions for some computational models and demonstrate their use in designing a simple vending machine. We also define an interaction relation between power and discrete domain to demonstrate a new methodology for estimating power dissipated by software. We observe that an interaction needs to be written only once. The discrete-power interaction is reused whether we are investigating power in a CMOS implementation or software implementation.

Numerous Rosetta analysis tools are under development. A Java simulator and a verifier will soon be available to provide formal verification support to Rosetta. Commercial test vector and test case generation tools are available for generating VHDL test vectors and will soon be available for C++. Prototype IP reuse and matching tools have been developed to aid designers in automatically determining if a component meets a collection of problem requirements. In support of these activities, Rosetta is currently undergoing standardization by the Accellera EDA standards organization.

References

[1] P. Alexander, D. Barton, and C. Kong. *Rosetta Usage Guide*. The University of Kansas / ITTC, 2335 Irving Hill Rd, Lawrence, KS, 2000.

[2] Perry Alexander and Cindy Kong. Rosetta: Semantic support for model-centered systems-level design. *IEEE Computer*, 34(11):64–70, November 2001.

[3] L. Allison. *A practical introduction to denotational semantics*. Number 23 in Cambridge Computer Science Texts. Cambridge University Press, 1986.

[4] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:184–235, 1994.

[5] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems, December 1996. `http://www.brics.dk/RS/96/58/`.

[6] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, Cesar Munoz, Sam Owre, Harald Rueb, John

Rushby, Vlad Rusu, Hassen Saidi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *Fifth NASA Langley Formal Methods Workshop*, Williamsburg, VA, June 2000. `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`.

[7] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, April 1994.

[8] J. R. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *Proceedings of the second International Conference on Application of Concurrency to System Design*, June 2001.

[9] Ernst Christen. The VHDL 1076.1 language for mixed-signal design. *EE Times*, June 1997. Analogy, Inc.

[10] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992. World Scientific Publishing Co.

[11] J. A. Goguen and R. M. Burstall. Introducing institutions. *Lecture Notes in Computer Science*, 164:221–255, 1984.

[12] James Gosling and Henry McGilton. *The Java Language Environment: A White Paper*. Sun Microsystems, Mountain View, CA, May 1996. `http://java.sun.com/docs/white/langenv/`.

[13] The UML Group. *UML Metamodel*. Rational Software Corporation, Santa Clara, CA, 1.1 edition, September 1997. http://www.rational.com.

[14] John Grundy. Aspect-oriented requirements engineering for component-based software systems. In *Proceedings of RE'99*, Limerick, Ireland, June 1999. IEEE.

[15] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, 1996.

[16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, 1997.

[18] C. Kirkwood and K. Norris. *Formal Description Techniques*, volume III, chapter Some Experiments using Term Rewriting Techniques for Concurrency, pages 527–530. Elsevier Science Publishers B.V., 1991.

[19] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *In Proceedings of WISP 2001*, Budapest, Hungary, May 2001.

[20] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[21] Edward A. Lee and Yuhong Xiong. System-level types for component-based design. Technical report, University of California at Berkeley, February 2000.

[22] Greg Nordstrom, Janos Sztipanovits, Gabor Karsai, and Akos Ledeczi. Metamodeling - rapid design and evolution of domain-specific modeling environments. In *Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based Systems*, Nashville, Tennessee, March 1998.

[23] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[24] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[25] Marie-Noelle Terrasse, Marinette Savonnet, and George Becker. An uml-metamodeling architecture for interoperability of information systems. In *4th International Conference on Information Systems Modelling*, Hradec nad Moravici, Czech Republic, May 2001.

[26] Pei-Hsin Ho Thomas A. Henzinger and Howard Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[27] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chen Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, pages 1–18, 1996. Kluwer Academic Publishers, Boston.

[28] Mark Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. PhD thesis, University of New South Wales, Kensington, Australia, October 1992.

[29] Pamela Zave. Feature-oriented description, formal methods, and dfc. In Stephen Gilmore and Mark Ryan, editors, *Language Constructs for Describing Features*, pages 11–26. Springer-Verlag London Ltd, 2000/2001. Feature Integration in Requirements Engineering.