Rosetta Functional Specification  Domains

Perry Alexander
EECS Department / ITTC
The University of Kansasas
What is Rosetta?

- Rosetta is a language for describing systems
  - Presently the focus is on complex electronic systems -> SOC
  - Being explored for complex mechanical systems
- Rosetta defines systems by writing and composing models
  - Each model is defined with respect to one domain
  - Composition provides definition from multiple perspectives
- Rosetta consists of a syntax *(a set of legal descriptions)* and a semantics *(a meaning associated with each description)*
• A Rosetta *domain* provides a vocabulary for model specification
  – Defines commonly used abstractions
  – Defines state and time

• A Rosetta *interaction* provides a definition of how specification domains interact
  – Defines when facts from one domain cause facts to be true in another
  – Causes information to cross domains when models are composed
Understanding Facet Definitions

- Facets provide mechanisms for defining models and grouping definitions

Facet Name: facet trigger(x::in real; y::out bit) is

Variables:
- s::bit;

Terms:
- begin continuous
- t1: s@t+1ns =
  - if s=1 then if x>=0.4 then 1 else 0 endif;
  - else if x=<0.7 then 0 else 1 endif;
- t2: y@t+10ns=s;
- end trigger;
The Logic Domain

- The logic domain provides a basic set of mathematical expressions, types and operations
  - Number and character types and operations
  - Boolean and bit types and operations
  - Compound types and operations
    » bunch, set, sequence, array
  - Aggregate types and operations
    » record, tuple
  - Function function and operation definition

- **Best thought of as the mathematics facet**
  - No temporal or state concepts
The State-Based Domain

- **The state–based domain supports defining behavior by referencing the current and next state**
- **Basic additions in the state–based domain include:**
  - $S$ – The state type
  - `next::[S→S]` – Relates the current state to the next state
  - $x@s$ – Value of $x$ in state $s$
  - $x'$ – Standard shorthand for $x@next(s)$
Defining State Based Specifications

- Define important elements that describe state
- Define properties in the current state that specify assumptions for correct operation
  - Frequently called a precondition
- Define properties in the next state that specify how the model changes its environment
  - Frequently called a postcondition
- Define properties that must hold for every state
  - Frequently called invariants
facet pp-function(inPulse:: in PulseType;
inPulseTime:: in time;
o:: out command) is
  use timeTypes; use pulseTypes;
pulseTime :: time;
pulse :: PulseType;
begin state-based
 L1: pulseTime >= 0;
 L2: pulse=A1 and inPulse=A2 => pulse’=none;
 L3: pulse=A1 and inPulse=A1 => pulse’=none and
     o’=interpret(pulseTime,inPulseTime);
end pp-function;
When to use the State-Based Domain

• **Use state-based specification when:**
  – When a generic input/output relation is known without details
  – When specifying software components

• **Do not use state-based specification when:**
  – Timing constraints and relationships are important
  – Composing specifications is anticipated
The Finite State domain

- The finite-state domain supports defining systems whose state space is known to be finite.
- The finite-state domain is a simple extension of the state-based domain where:
  - $S$ is defined to be or is provably finite.
**Trigger Example**

- There are two states representing the current output value
  - \( S::type = 0++1; \)
- The next state is determined by the input and the current state
  - \( L1: \text{next}(0) = \text{if } i \geq 0.7 \text{ then } 1 \text{ else } 0 \text{ endif; } \)
  - \( L2: \text{next}(1) = \text{if } i < 0.3 \text{ then } 0 \text{ else } 1 \text{ endif; } \)
- The output is the state
  - \( L3: o' = s; \)
facet trigger(i:: in real; o:: out bit) is
   S::type = 0++1;
begin state-based
   L1: next(0) = if i>=0.7 then 1 else 0 endif;
   L2: next(1) = if i=<0.3 then 0 else 1 endif;
   L3: o'=s;
end trigger;
When to use the Finite State Domain

- **Use the finite-state domain when:**
  - Specifying simple sequential machines
  - When it is helpful to enumerate the state space

- **Do not use the finite-state domain when**
  - The state space cannot be proved finite
  - Usage over specifies the properties of states and the next state function
The Infinite State Domain

- **The infinite–state domain** supports defining systems whose state spaces are infinite.
- **The infinite–state domain** is an extension to the state-based domain and adds the following axiom:
  - $\text{next}(s) > s$
- **The infinite–state domain** asserts a total ordering on the state space:
  - A state can never be revisited.
The Pulse Processor Revisited

• The initial pulse arrival time must be greater than zero
  – \( L1: \text{pulseTime} \geq 0; \)

• Adding the infinite state restriction assures that time advances

• If the initial pulse is of type A1 and the arriving pulse is of type A2, reset and wait for another pulse
  – \( L2: \text{pulse}=A1 \text{ and inPulse}=A2 \text{ implies pulse’}=\text{none} \)

• If the initial pulse is of type A1 and the arriving pulse is of type A1, then output command
  – \( L3: \text{pulse}=A1 \text{ and inPulse}=A1 \text{ implies pulse’}=\text{none} \text{ and o’}=\text{interpret(pulseTime,inPulseTime)}; \)
• The discrete–time domain supports defining systems in discrete time
• The discrete–time domain is a special case of the infinite–state domain with the following definition
  – next(t)=t+\delta;
• The constant \( \delta \geq 0 \) defines a single time step
• The state type \( T \) is the set of all multiples of \( \delta \)
• All other definitions remain the same
  – next(t) satisfies next(t)>t
facet pp-function(inPulse::in PulseType; 
o::out command) is

use pulseTypes;
pulseTime :: T;
pulse :: PulseType;
begin discrete-time
L2: pulse=A1 and inPulse=A2 => pulse@t+delta=none;
L3:pulse=A1 and inPulse=A1 => pulse@t+delta=none and
    o@t+2*delta=interpret(pulseTime,t);
end pp-function;
Discrete Time Pulse Processor

- **State is the last pulse received and its arrival time or none**
- **The initial pulse arrival time must be greater than zero**
  - Guaranteed by definition of time
- **If the initial pulse is of type A1 and the arriving pulse is of type A2, reset and wait for another pulse**
  - L2: \( \text{pulse}=A1 \) and \( \text{inPulse}=A2 \) implies \( \text{pulse}_t+\text{delta}=\text{none} \)
- **If the initial pulse is of type A1 and the arriving pulse if of type A1, then output command in under 2 time quanta**
  - L3: \( \text{pulse}=A1 \) and \( \text{inPulse}=A1 \) implies \( \text{pulse}_t+\text{delta}=\text{none} \) and \( o_{t+2*\text{delta}}=\text{interpret}(\text{pulseTime},t) \);
- **No state should ever have a negative time value**
  - Guaranteed by the definition of time
When to use the Discrete Time Domain

• **Use the discrete–time domain when:**
  – Specifying discrete time digital systems
  – Specifying concrete instances of systems level specifications

• **Do not use the discrete–time domain when:**
  – Timing is not an issue
  – More general state-based specifications work equally well
The Continuous Time Domain

- **The continuous-time domain supports defining systems in continuous time**
- **The continuous-time domain has no notion of next state**
  - The time value is continuous – no next function
  - The “@” operation is still defined
    » Alternatively define functions over t in the canonical fashion
- **Derivative, indefinite and definite integrals are available**
Continuous Time Pulse Processor

- Not particular interesting or different from the discrete time version
  - Can reference arbitrary time values
  - Cannot use the next function
  - No reference to discrete time – must know what delta is
Continuous Time Pulse Processor

facet pp-function(inPulse::in PulseType; o::out command) is

  use pulseTypes;
  pulseTime :: T;
  pulse :: PulseType;
begin discrete-time
  L2: pulse=A1 and inPulse=A2 => pulse@t+5ms=none;
  L3: pulse=A1 and inPulse=A1 => pulse@t+5ms=none and o@t+10ms=interpret(pulseTime,t);
end pp-function;
Discrete time references are replaced by absolute time references with respect to the current time.

- Using 5ms and 10ms intervals rather than the fixed time quanta.
Using the Continuous Time Domain

• **Use the continuous–time domain when**
  – Arbitrary time values must be specified
  – Describing analog, continuous time subsystems

• **Do not use the continuous–time domain when:**
  – Describing discrete time systems
  – State based specifications would be more appropriate
Specialized Domain Extensions

- The domain **mechanical** is a special extension of the logic and continuous time domains for specifying mechanical systems.
- The domain **constraints** is a special extension of the logic domain for specifying performance constraints.
- Other extensions of domains are anticipated to represent:
  - New specification styles
  - New specification domains such as optical and MEMS subsystems
Domains and Interactions

- **Monotonic Logic**
  - State-Based
  - Continuous Time
  - Finite State
  - Infinite State
  - Discrete Time

- **Example Requirements definition domains and standard interactions**
  - Solid lines represent homomorphisms
  - Dashed lines represent incomplete interactions
More Information?

- The new Rosetta web page is available at:
  http://www.ittc.ukans.edu/Projects/SLDG/rosetta
- Email the authors at:
  alex@ittc.ukans.edu
dlb@averstar.com
- Come to the tutorial yesterday!
  - Slides will be available via the web page