Modular Semantics for Model-Oriented Design

Cindy Kong
ckong@ittc.ku.edu

The Information Technology and Telecommunication Center
The University of Kansas
WELCOME
&
ACKNOWLEDGEMENTS
Problem Statement

“Different paradigms can give quite different views of the nature of computation and communication. In a large system, different subsystems can often be more naturally designed and understood using different models of computation.” [Burch et al.]

Integration of different paradigms within one specification framework dictates:

- Common syntax (domain of discourse)
- Formal semantics that provides notion of consistency
- Translation of specifications
- Composition of specifications
Proposed Solution

- Formal semantics
  - Institution
    - Relates syntax to semantics
    - Defines notion of models satisfying a specification
    - Defines a logical system, e.g. equational reasoning, first-order logic, ...
    - Provides basis for sound and complete deduction calculus
  - Modularity in using several institutions
- Multi-model of computation framework
  - Identify unifying semantic domains (units of semantics)
    - Static
    - State-based
    - Trace-based
  - Define models of computation
    - State-based: continuous, discrete, finite-state
    - Trace-based: csp-trace
Key Contributions

- Definition of a formal semantics, giving an entailment system that allows reasoning over correctness of a heterogeneous design
- Definition of multiple unifying semantic domains and models of computations within one framework
- Definition of relations between specifications
- Demonstration of composition of specifications
- Demonstration of new heterogeneous design methodology
- Demonstration of re-use of domain-specific views
Overview

- Preliminaries
- Modular semantics
  - Static semantics
  - State-based semantics
    - Hidden algebras
    - Coalgebras
  - Trace-based semantics
- Specification in the Rosetta Language
  - Units of semantics
  - Models of computation
- Examples and Application
  - Hybrid system
- Related work and future work

9/15/2004
PRELIMINARIES
Category Theory

- **Category $\mathcal{C}$**
  - Collection of objects $|\mathcal{C}|$
  - Collection of arrows $||\mathcal{C}||$ (with $\text{dom}$ and $\text{cod}$)
  - Composition of arrows
  - Identity arrow for each object

- **Examples**
  - Category of algebras
    - The objects are algebras
    - The arrows are homomorphisms between algebras
  - Category of sets
    - The objects are sets
    - The arrows are functions
Concrete Example

sort Bit;
operations
  zero: -> Bit;
  one: -> Bit;
  succ: Bit -> Bit;
equations
  succ(zero) = one;
  succ(one) = zero;

Functor \( d: \text{D} \rightarrow \text{C} \) is called a diagram of shape \( \text{D} \) in category \( \text{C} \).
Institution Theory

- Formalizes:
  
  *Truth is invariant under changes of notation*

- Institution \((\text{Sign}, \text{Mod}, \text{Sen}, \models)\)
  
  - **Sign**: category of signatures
  
  - **Sen**: \(\text{Sign} \rightarrow \text{Set}\) functor giving set of sentences for each signature
  
  - **Mod**: \(\text{Sign} \rightarrow \text{Cat}^{\text{op}}\) functor giving category of models for each signature
  
  - \(\models_{\Sigma} \subseteq \text{Mod}(\Sigma) \times \text{Sen}(\Sigma)\) signature-indexed family of satisfaction relations such that for
    \[
    (\phi: \Sigma \rightarrow \Sigma') \in \| \text{Sign} \|, e \in \text{Sen}(\Sigma), M' \in \text{Mod}(\Sigma') \\
    M' \models_{\Sigma'} \text{Sen}(\phi)(e) \text{ if and only if } \text{Mod}(\phi)(M') \models_{\Sigma} e
    \]
MODULAR SEMANTICS
Static Semantics – Programming in the small

- Notion of fixed data
- Notion of invariance
- Signature \((S_{Stc}, \Sigma_{Stc})\)
  - \(S_{Stc}\) set of sorts
  - \(\Sigma_{Stc}\) set of operators \(S^*_{Stc} \times S_{Stc}\)
- Algebra
  - \(S_{Stc}\) – indexed family of non-empty sets, carriers \(A_{Stc}\)
  - \(S^*_{Stc} \times S_{Stc}\) – indexed family of maps
    \[
    \alpha_{u,s} : \Sigma_{Stc_{u,s}} \rightarrow [A_{Stc_{u}} \rightarrow A_{Stc_{s}}]
    \]
- Algebra morphism from \(\langle A_{Stc}, \alpha \rangle \rightarrow \langle A'_{Stc}, \alpha' \rangle\) is map
  \[f : A_{Stc} \rightarrow A'_{Stc}\] such that
  \[f(\alpha(\sigma)(a_1, \ldots, a_n)) = \alpha'(\sigma)(f_{s_1}(a_1), \ldots, f_{s_n}(a_n))\]
- Equation \((\forall X)t1 = t2\)
Static Semantics – Programming in the large

- Specification is \((S_{Stc}, \Sigma_{Stc}, E_{Stc})\)
- Algebra \(A_{Stc}\) satisfying equation \(e\) iff \(a^*(t1) = a^*(t2)\) for every assignment \(a : X \rightarrow |A_{Stc}|\),
  \[ A_{Stc} \models_{\Sigma_{Stc}} e \]
- Institution for static algebras (equational-\([Goguen]\)) \((\text{Sig}_{Stc}, \text{Alg}_{Stc}, \text{Eqn}_{Stc}, \models_{Stc})\)
  - \(\text{Sig}_{Stc}\) category of static signatures and morphisms
  - \(\text{Alg}_{Stc}\) functor giving category of static algebras for each signature
  - \(\text{Eqn}_{Stc}\) functor giving a set of equations for each signature
  - \(\models_{Stc}\) satisfaction such that
    \[ A'_{Stc} \models_{\Sigma_{Stc}} \varphi(e) \iff A'_{Stc} \models_{\Sigma_{Stc}} e \text{ with } \varphi : \Sigma_{Stc} \rightarrow \Sigma'_{Stc} \]
Static Semantics – Specification construction

- Specification extension
  - Extension satisfies *no confusion* and *no junk* constraint
  - \((S'_\text{Stc}, \Sigma'_\text{Stc}, E')\) *extends* \((S_{\text{Stc}}, \Sigma_{\text{Stc}}, E)\) \(\Rightarrow S_{\text{Stc}} \subseteq S'_{\text{Stc}}, \Sigma_{\text{Stc}} \subseteq \Sigma'_{\text{Stc}}, E \subseteq E'\)
  - Extension is an inclusion morphism, more specifically it is an enrichment signature morphism that is conservative

- Specification parameterization and instantiation
  - Parameterization – defines properties over a class of specifications
  - Instantiation – reduces class to a particular specification, and involves binding signature morphism

- Specification inclusion
  - Allows information hiding that involves a signature inclusion along with an information hiding operator \((\ )\)

- Specification use
  - Use packages

- Specification composition
  - Pushout of two specifications – syntactic composition

9/15/2004
State-based Semantics – Programming in the small

- Notion of observing a current state and change of observations over a next transformation function
  - A state is only identified by its attributes
  - Two states that have same attributes are undistinguishable and are said to be behaviorally equivalent

- State-based signature \((S_{SB}, \Sigma_{SB})\)
  - \(S_{SB} = (\text{State}, S_v)\)
  - \(\Sigma_{SB} = (\text{isInit}, Y, \text{next}, \Phi, \Omega, \Delta)\)
    - \(Y\) set of generalized hidden constants \(\text{cst} : S_{V_{0,\ldots,n}} \to \text{State}\)
    - \(\Phi\) optional set of operations \(\phi : \text{State} \times S_{V_{0,\ldots,n}} \to \text{State}\)
    - \(\Omega\) set of attributes \(\omega : \text{State} \times S_{V_{0,\ldots,n}} \to S_v\)
    - \(\Delta\) set of data operations \(\delta : S_{V_{0,\ldots,n}} \to S_v\)
    - Distinction between operators of \(\text{next}\) and \(\text{next}\)
State-based Semantics – Programming in the small

- A state-based signature: hidden signature [Goguen]
  - Hidden sort = State
  - Visible data universe = \((S_r, \Delta, D_{SB})\)
  - At most one hidden sort occurs in \(Y\) or \(\Omega\)

- Behavioral Satisfaction
  - A context of sort \(h\) is a visible sorted \(\Sigma\)-term that has a single occurrence of a new variable symbol \(z\) of sort \(h\), e.g. \(x(z), x(\text{next}(z))\).
  - A hidden algebra behaviorally satisfies equation \(e\)
    
    \[
    A \models_\Sigma (\forall X) t = t' \quad \text{if} \quad t_1 = t'_1, \ldots, t_m = t'_m
    \]

    iff for each appropriate context \(c\) and assignment \(\theta : X \rightarrow A\)

    \[
    \theta^*(c[t]) = \theta^*(c[t'])
    \]

    whenever \(\theta^*(c_j[t_j]) = \theta^*(c_j[t'_j])\) for \(j = 1, \ldots, m\) and all appropriate \(c\)
State-based Semantics – Programming in the small

- State-based specification \((S_{SB}, \Sigma_{SB}, E)\)
  - \((S_{SB}, \Sigma_{SB})\) is a state-based signature
  - \(E = E_\Delta \oplus E_\Omega\) disjoint union of 2 sets of equations
  - Induces a hidden specification \((State, \Sigma_{SB}, E_\Omega)\)

- Consistency of state-based specification
  - Consistent iff induced hidden specification has a model with non-empty carriers and all equations \(E_\Delta\) are consistent
  - Necessary condition: \(E\) is D-safe
  - Sufficient condition: locality of equations
    - Local equation: local terms and conditions are visibly sorted and use only \(\Psi\)-operations
    - Local term: every proper subterm is a \(\Psi\)-subterm
  - Non-local: use rewriting and provide a model
State-based Semantics – Programming in the large

- State-based signature morphism
  - Hidden signature morphism
  - Identity over the visible data \((V, \Psi)\)
  - Maps hidden sorts to hidden sorts
    \[
    \text{morphism} \quad (S_{SB}, \Sigma_{SB}) \rightarrow (S'_{SB}, \Sigma'_{SB})
    \]
    \[
    \text{signature morphism} \quad \varphi : \Sigma_{SB} \rightarrow \Sigma'_{SB}
    \]
    \[
    \text{if} \quad \sigma' \in \Phi' \quad \text{or} \quad \sigma' \in \Omega' \quad \text{then} \quad \exists \quad \sigma \in \Phi \quad \text{or} \quad \sigma \in \Omega \mid \sigma' = \varphi(\sigma)
    \]
- Sub-system morphism instead of enrichment morphism
- Only one State sort, use of qualified name through a renaming morphism to distinguish between State sort of different specifications
State-based Semantics – Programming in the large

- Institution for state-based algebras
  - Category of state-based signature and morphisms
    \[ \text{Sign}_{SB} \]
  - Functor giving a set of equations for each signature
    \[ \text{Sen}_{SB} \]
  - Functor giving a category of hidden algebras for each signature
    \[ \text{Mod}_{SB} \]
  - Satisfaction relation
    \[ \models_{SB} \]
  - Satisfaction condition
    \[ A' \models_{SB} e \quad \text{iff} \quad A' \models_{SB} \phi(e) \]
State-based Semantics – Coalgebras

- Cirstea’s work: Hidden algebras → Coalgebras
- State-based signature → destructor hidden signature (by leaving out \( Y \) and \( \Phi \)) → abstract cosignature

\[
(Set_{DSB}^{SB}, F_{\Sigma_{SB}}) \quad \text{with} \quad F_{\Sigma_{SB}} : Set_{DSB}^{SB} \to Set_{DSB}^{SB} \\
(X_{S_1}, \ldots, X_{S_n}, X_{State}) \to (X_{S_1}, \ldots, X_{S_n}, \prod_{k \in 1, \ldots, l} X_{S_k}^{X_{S_0}, \ldots, n} \times X_{State}^{X_{S_0}, \ldots, n})
\]

- Example:
  - State-based signature \( \text{State, Natural} \)
    \( s_0 :\to \text{State, } x : \text{State} \to \text{Natural, } \text{next : State} \to \text{State, } \Delta_{\text{Natural}} \)
  - Destructor hidden subsignature
    \( (\{\text{Natural, State}\}, \{x : \text{State} \to \text{Natural, next : State} \to \text{State}\} \cup \Delta_{\text{Natural}}) \)
  - Associated abstract cosignature
    \( (Set_{N}^{\{\text{Natural, State}\}}, F) \quad \text{with} \quad FX_{State} = N \times X_{State} \)
  - A coalgebraic structure
    \( \alpha : X_{State} \to N \times X_{State} \)

9/15/2004
State-based Semantics – Specification construction

- **Extension:** similar in essence to static specification extension
  - The signature morphism is reverse
    \[(S_{SB}, \Sigma_{SB}, E) \xrightarrow{c} (S'_{SB}, \Sigma'_{SB}, E') \text{ iff } \exists \varphi : (S'_{SB}, \Sigma'_{SB}) \rightarrow (S_{SB}, \Sigma_{SB})\]
- **Parameterization:**
  - 3 parameter modes: input, output and design
- **Instantiation:** may involve state dependent bindings of parameters
- **Translation:** mapping of properties of the State sort from one specification to another
- **Inclusion:** similar to static inclusion, but may be supplemented by a translation relating states of specifications involved in inclusion
- **Use:** as for static. In this work, all packages are static
State-based Semantics – Specification composition

- Category of state-based specifications as objects and extensions as arrows
- Composition uses categorical notion of colimit
- Composition of two specifications sharing a common parent through a pushout
- Composition of two specifications on different subtrees, translation may first be needed

```
SpecShared

Spec1   Spec2
/     \        \------ OtherSpec2
|      |         |
|      |         |
|      |         |
|      |         |
|      |         |
Spec3 = Spec1 + Spec2
```
Trace-based semantics

- Notion of traces and operations over traces to model computation runs
- Equational signature
- Same semantics as for static
  - Institution of equational reasoning
- Enforcement of a Trace(T) sort
- Available Operations: head, tail, add, sequence, interleave, restriction, order, ...
Specification Construction across Semantic Domains

- Conservative extension from static to state-based and from static to trace-based
- Institution morphism from static to state-based is strong, persistent and additive similar to CafeOBJ’s institution morphism
- Specification translation from static to state-based
  - Static represents data and invariant properties in a state-based specification
  - Minimal representation:
    \[ Spec_{SB} = (S_{Spec_{Stc}} \cup \{State\}, \Sigma_{Stc} \cup next, E_{Stc} \cup E_{SB}) \]
- Specification translation from state-based to static described by Goguen et al.
  - Translation of behavioral specification into ordinary algebraic specification
Specification Translation from State-based to Trace-based

- One-way translation $Spec_{SB} \rightarrow Spec_{TB}$
- For each input $I$ in $Spec_{SB}$, an input set of traces of type of $I$ in $Spec_{TB}$
- Same for output parameters
- All declarations of $Spec_{SB}$ become declarations of $Spec_{TB}$
- Add declarations of:
  - A variable $T_s :: Trace(State)$ representing set of traces of all reachable states
  - A variable someTrace representing a trace
  - A variable $n$ of sort natural used as position of state in trace
- All equations of $Spec_{SB}$ are included in $Spec_{TB}$
- Add 2 new equations: state_def - equating State to actual, and newT - stating
  
  $\text{someTrace} \in T_s, s \in State \text{ such that } \text{someTrace}[n] = s$
  
  and $\text{next}(s, I_0[n], \ldots, I_k[n]) = \text{someTrace}[n+1]$
SPECIFICATION IN ROSETTA
The Domain organization

Static (prelude)

- state-based
  - continuous
  - discrete
    - continuous-time
    - discrete-time
    - frequency
    - finite-state
  - trace-csp
    - trace-csp

- trace-based
  - RF
  - digital
  - sequential-machine
  - synchronous

Unit of Semantics

Model of Computation

Engineering Modeling
Static Modeling

- Semantics given by the previously defined static (equational) semantics
- Specification
  - Defines a number of types: Universal, Element, Number, Complex, Real, …, Function, Set, Sequence, …
  - Defines a number of operators over each sort
- Static domain

Static domain semantics (Boolean)

\[ S_{Stc} = \{\ldots, \text{Boolean}, \ldots\} \]

\[ \Sigma_{Stc} = \{\ldots, \text{false} : \rightarrow \text{Boolean}, \text{true} : \rightarrow \text{Boolean}, \text{not} : \text{Boolean} \rightarrow \text{Boolean}, \ldots \]

\[ \ldots, \text{or} : \text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean}, \ldots\} \]
domain static::null is
// -------------------------------------------------------------
// Boolean types
// -------------------------------------------------------------
Boolean :: type is enumeration (false, true);
// -------------------------------------------------------------
// Functions for boolean type
// -------------------------------------------------------------

not__(R :: Boolean ) :: Boolean;
__or__ ( L, R :: Boolean ) :: Boolean;

begin
not_false: (not false) = true;
not_true: (not true) = false;
true_or_true: (true or true) = true;
true_or_false: (true or false) = true;
false_or_true: (false or true) = true;
false_or_false: (false or false) = false;
end domain static;
Initial Algebra for Static

Boolean

'false'

'true'

'n'ot false'

'n'ot true'

'false or false'

'false or true'

N

...''0'' ...

...''1'' ...

'succ(0)'

...
State-based Modeling

- State-based semantics
  - Institutions of Hidden Algebras, Coalgebras
- Specification
  - State type
  - Next function that takes a state and a number of inputs and returns a new state
  - Extends static domain
- State-based domain semantics
  \[ S_{SB} = (\text{State}, S_{Stc}) \]
  \[ \Sigma_{SB} = (\text{isInit}, Y_{SB}, next, \{} ,\{} ,\{} ,\{} ,\{__@__\} \cup \Sigma_{Stc} ) \]
- Coalgebras
  \[ |A|_{State} \xrightarrow{\gamma_{\text{next}}} \{\} \cup |A|_{State} \]
  \[ |A|_{State} \xrightarrow{\zeta} |A|_{State} \]
**State-based Domain Specification**

```plaintext
domain state_based(State::design Type) :: static is

s :: State;
next:: Function;
__@__[T::Type](lhs::*(st::State) -> T *); rhs::State)::T is lhs(rhs);
isInit(s::State)::Boolean;

begin
   // next: State x Si ... x Sn -> State with Si,...,Sn: one or more types
   return_type_next: ret(next) = State;
   domain_next: dom(next) = State;
end domain state_based;
```
The Discrete Domain Specification

```plaintext
domain discrete(DiscState::design Type) :: state_based(DiscState) is

  isDiscrete(DiscreteSet::Type)::Boolean =
      exists (fnc::*(st::DiscreteSet)::Integer*> |
        forall(s1,s2::DiscreteSet|)
          (s1 /= s2) => (fnc(s1) /= fnc(s2))));

begin

  discrete_attributes: forall (fnc::getAttributes() | isDiscrete(ran(fnc)));

end domain discrete;
```
Finite-state \(\Rightarrow\) observations are finite and discrete

The size of the set is a natural number

\[\text{size} = 4\]

```plaintext
domain finite_state(FState::design Type) :: discrete(FState) is
  isFinite(FiniteSet::Type)::Boolean is
    #FiniteSet in Natural;
begin
  fs1:forall (fnc::getAttributes() | isFinite(ran(fnc)));
end domain finite_state;
```
The Continuous Domain

Continuous observation of states $\Rightarrow$ all observations have continuous variations with respect to a continuous observation of states

\[ \frac{\Delta f}{\Delta s} = \frac{f(next(s)) - f(s)}{contAttr(next(s)) - contAttr(s)} \]

```plaintext
domain continuous :: state_based is
    contAttr(st::State)::Real;
    variation[T::Type](fnc::*st::State::*T>;st::State;next_st::State::*T is
        (f(next_st) - f(st)) / (contAttr(next_st)-contAttr(st));

begin
end domain continuous;
```
Trace-based Modeling

- Semantics
  - Static semantics (institution of equational logic)
  - As traces represent computation runs, can use coalgebras as models as well

- Specification
  - Notion of traces
  - Operations as defined in trace semantics
  - Extends static domain
domain trace_based()::static is
Trace(T::Type)::Type;
emptyTrace::Trace(Universal) is constant;
add[Event::Type](tr::Trace(Event);ev::Event)::Trace(Event);
head[Event::Type](tr::Trace(Event))::Event;
tail[Event::Type](tr::Trace(Event))::Trace(Event);

isEmpty[Event::Type](tr::Trace(Event))::Boolean is
tr = emptyTrace;

getEventAt[Event::Type](tr::Trace(Event);pos::Natural)::Event is
if (not isEmpty(tr))
   else if (pos = 0) then head(tr)
      else getEventAt(tail(tr),pos-1)
   end if;
end if;

...
Examples and Applications
Example of a Stack Datatype

```plaintext
facet stackDT::static is
    Stack::type;
    emptyStack::Stack is constant;
    push(stcParam::Stack; n::Natural)::Stack;
    pop(stcParam::Stack)::Stack;
    top(stcParam::Stack)::Natural;
    val::Natural;
    stcVar::Stack;
begin
    pop_empty: pop(emptyStack) = emptyStack;
    top_empty: top(emptyStack) = 0;
    pop_push: pop(push(val, stcVar)) = stcVar;
    top_push: top(push(val, stcVar)) = val;
end facet stackDT;

S_{stackDT} = S_{Stc} \cup \{Stack\}
\Sigma_{stackDT} = \Sigma_{Stc} \cup \{emptyStack, push, pop, top\}
E_{stackDT} = E_{Stc} \cup \{pop_empty, top_empty, pop_push, top_push\}
```

9/15/2004
Initial algebra for stackDT

\[ \theta_{\text{Natural}}(val) = 0 \]
\[ \theta_{\text{Stack}}(stc) = \text{emptyStack} \]
Isomorphism between $N_{\text{stackDT}}$ and $N_{\text{Stc}}$

$N_{\text{stackDT}} | \phi = N_{\text{stackDT}}$

$N_{\text{stackDT}} | \phi$ satisfies static and is isomorphic to $N_{\text{static}}$
Composition of State-based Parameterized Specifications

StateSet::Type;
memNext(st::State;val::Natural)::State;

facet memoryA(val::input Natural)
    ::discrete(StateSet) is
    memA(st::State)::Natural;
begin
    initA: isInit(s) => memA@s = 0;
    next_def: next = memNext;
    lA: memA@next(s,val) = val;
end facet memoryA;

facet memoryB(val::input Natural)
    ::discrete(StateSet) is
    memB(st::State)::Natural
begin
    initB: isInit(s) => memB@s = 0;
    next_def: next = memNext;
    lB: memB@next(s,val) = val+memB;
end facet memoryB;

facet twoMemory(val::input Natural)::discrete(StateSet) is
    memoryA(val) + memoryB(val);
Composition of Parameterized Specifications

Discrete(StateSet)
next::Function

memoryA(val)
memA::State → Natural
next(st::State;val::Natural)::State

memoryB(val)
memB::State → Natural
next(st::State;val::Natural)::State

twoMemory(val)
memA::State → Natural
memB::State → Natural
next(st::State;val::Natural)::State

Pullback of Signature Morphisms
Composition of Parameterized Specifications

\[ \gamma_{\text{discrete}} = \text{next} \]

\[ \gamma_{\text{memoryA}} = (\text{memA, next}) \]

\[ \gamma_{\text{memoryB}} = (\text{memB, next}) \]

\[ \gamma_{\text{twoMemory}} = (\text{memA, memB, next}) \]

Pushout of Coalgebras
Trace-based Memory A Specification

\[
\begin{align*}
\text{StateSet} & \text{: Type;} \\
\text{memNext}(\text{st}:: \text{State}; \text{val}:: \text{Natural}) & \text{: State;} \\
\text{facet} \quad \text{traceMemA}(\text{val}:: \text{input Trace(Natural)}) & \text{: trace_based()} \text{ is} \\
\text{memA}(\text{st}:: \text{State}) & \text{: Natural;} \\
\text{StateTrace} & \text{: Trace(State);} \\
\text{someTrace} & \text{: StateTrace;} \\
\text{s}:: \text{State}; \text{ next}:: \text{Function}; \quad \ldots \quad \text{// All declarations from domains} \\
\text{pos}:: \text{Natural;} \\
\text{begin} \\
\text{initA: isInit(s) } & \Rightarrow ((\text{memA(s)} = 0) \text{ and } (\text{pos} = 0)); \\
\text{next_def: next} & = \text{memNext}; \\
\text{lA: memA(next(s, getEventAt(val, pos)))} & = \text{getEventAt(val, pos);} \\
\text{newT1: getEventAt(someTrace, pos)} & = \text{s;} \\
\text{newT2: next(s, getEventAt(val, pos))} & = \text{getEventAt(someTrace, pos+1);} \\
\text{end facet} \quad \text{traceMemA;} \\
\end{align*}
\]

\[\gamma_{\text{traceMemA}} \equiv (\text{memA(head), tail})\]
Specification of a Hybrid Automaton

- Hybrid automaton [Henzinger]
  - Variables: $x$, dotted $x$ ($\dot{x}$), $x'$
  - Control graph $(V,E)$ of control modes and edges
  - Predicates:
    - Initial
    - Invariant
    - Flow conditions: predicate for continuous change
    - Jump conditions: predicate for each control switch
  - Events over control switches (events)
Two states for the heater: on or off

Continuous variation of the temperature: \( x \)

- heater on => temperature \( x \) increases at rate of \( 5 - 0.1x \) per minute
- heater off => temperature \( x \) decreases at rate of \( -0.1x \) per minute
The Heater Specification

\textbf{facet} heater(x::input Real; ctrl::output ControlMode):: finite_state is

\begin{verbatim}
mode(s::State)::ControlMode;

begin

  initial: isInit(s) => (mode@s = off);

  next_def: next = *(st::State;x::Real)::State*;

  output: ctrl = mode@s;

  off_to_on: ((mode@s = off) and (x =< 18)) => (mode@next(s,x) = on);

  on_to_off: ((mode@s = on) and (x >= 22)) => (mode@next(s,x) = off);

  off_to_off: ((mode@s = off) and (x >= 19)) => (mode@next(s,x) = off);

  on_to_on: ((mode@s = on) and (x =< 21)) => (mode@next(s,x) = on);

  grey_area_off: ((x < 19) and (x > 18) and (mode@s = off)) =>
  ((mode@next(s,x) = off) xor (mode@next(s,x) = on));

  grey_area_on: ((x > 21) and (x < 22) and (mode@s = on)) =>
  ((mode@next(s,x) = off) xor (mode@next(s,x) = on));

end facet heater;
\end{verbatim}
The Temperature Specification

```plaintext
cfacet temperatureVariation(ctrl::input ControlMode; x::output Real):: continuous is
temp(s::State)::Real;
begin
  initial: isInit(s) => ((temp@s = 20) and (contAttr@s = 0));
  next_def: next = <*(st::State;ctrl::ControlMode)::State*>;
  mono_increase: contAttr@next(s,ctrl) > contAttr@s;
  output: x = temp@s;
  off_cool: (ctrl = off) =>
    (variation(temp,s,next(s,ctrl)) = -0.1 * temp@s);
  on_heat: (ctrl = on) =>
    (variation(temp,s,next(s,ctrl)) = 5 - 0.1 * temp@s);
  next_heat: temp@next(s,ctrl) = temp@s +
    variation(temp,s,next(s,ctrl)) *
    (contAttr@next(s,ctrl)) - contAttr(s));
end facet temperatureVariation;
```
facet thermostat():: state_based is
ctrl(st::State)::ControlMode;
x(st::State)::Real;
begin
next_def: next = <*(st::State)::State*>;
heater_comp: heater(x@s, ctrl@s);
temperature_comp: temperatureVariation(ctrl@s, x@s);
inv_off: (ctrl@s = off) => (x@s >= 18);
inv_on: (ctrl@s = on) => (x@s <= 22);
end facet thermostat;
Analysis of the Thermostat Specification

- Two observations of the state
- The values of each observation provided by Heater or by Temperature Variation specifications
- Models that satisfy Thermostat will have (minimal) states as pairs \((\text{controlmode}, \text{temp})\) with \(\text{controlmode}=\text{ctrl}(s)\) and \(\text{temp}=\text{x}(s)\)
- \text{Controlmode}: on or off
- \text{Temp}: a real number between 18 and 22
- If considering discrete Thermostat models, \text{temp} will have discretized values through “sampling”
RELATED WORK
AND
FUTURE WORK
Related Work


- Ptolemy II - *Heterogeneous Concurrent Modeling and Design in Java* – J. Davis, C. Hylands, B. Kienhuis, E. Lee, et al.; University of California at Berkeley

Related Work

- Feature Engineering – Feature-Oriented Description, Formal Methods, and DFC – P. Zave
- Aspect-oriented –
  - Aspect-Oriented Programming – G. Kiczales et al.
  - Aspect-Oriented Requirements Engineering for Component-Based Software Systems – J. Grundy
Related Work


Related Work

- A Framework for Multi-Notation Requirements Specification and Analysis – N. Day and J. Joyce

- Constructing Multi-Formalism State-Space Analysis Tools: Using rules to specify dynamic semantics of models – M. Pezze and M. Young

- A Multi-Formalism Specification Environment – E. Ipser, Jr and D. Wile

- Acme: An Architecture Description Interchange Language – D. Garlan, R. Monroe and D. Wile
Conclusion

- Modular formal semantics
- Framework supporting different models of computation
- Future Work
  - Extension of semantics to order sorted institution
  - Definition of engineering domains: definition of units of measurement, definition of engineering formulas.
  - Automatic verification tool