rosetta

Reducing the cost of systems-level information

Modular Semantics for Model-Oriented Design

Cindy Kong
ckong@ittc.ku.edu

The Information Technology and Telecommunication Center
The University of Kansas

Information and Telecommunication Technology Center

KU

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

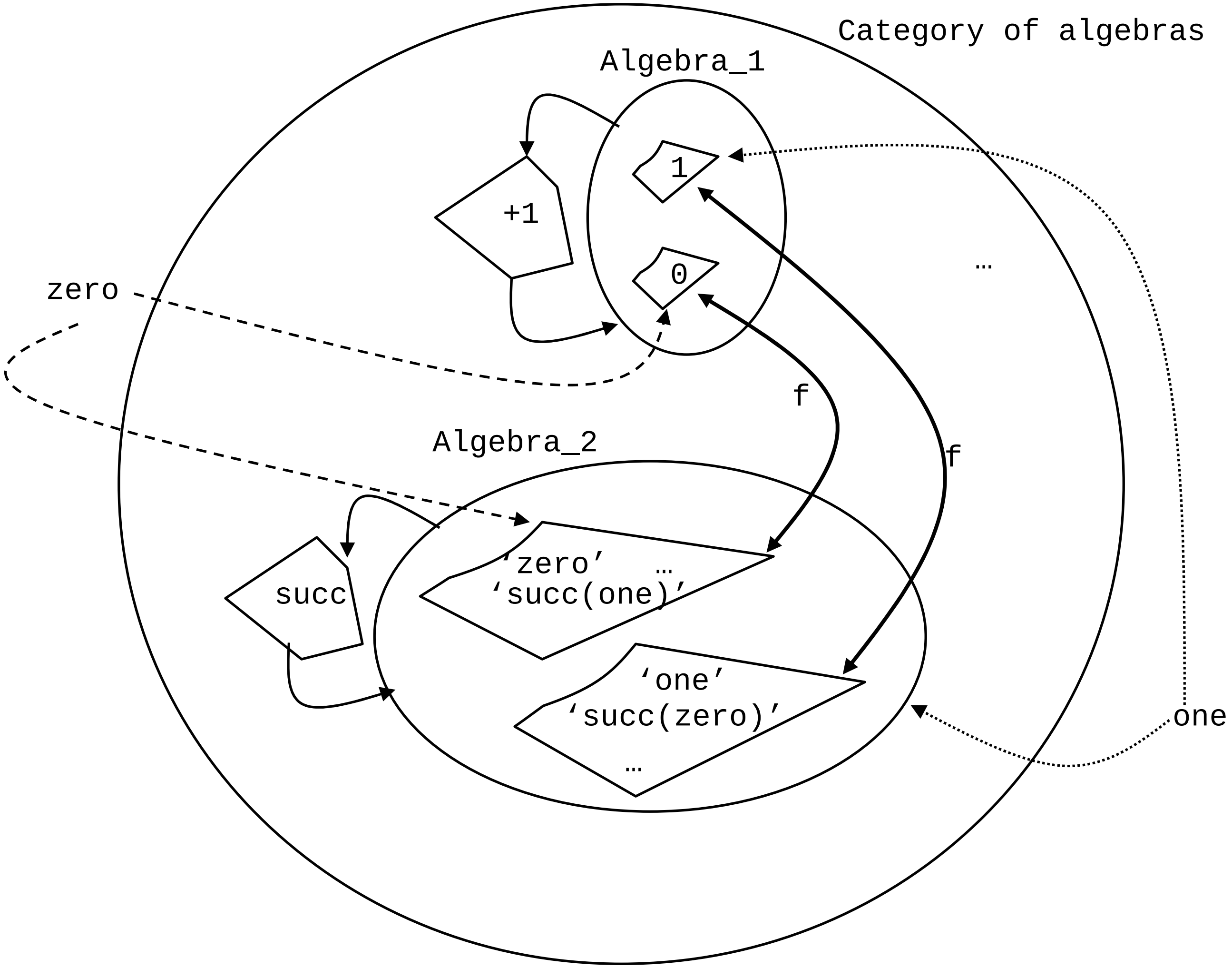
PRELIMINARIES

Category Theory

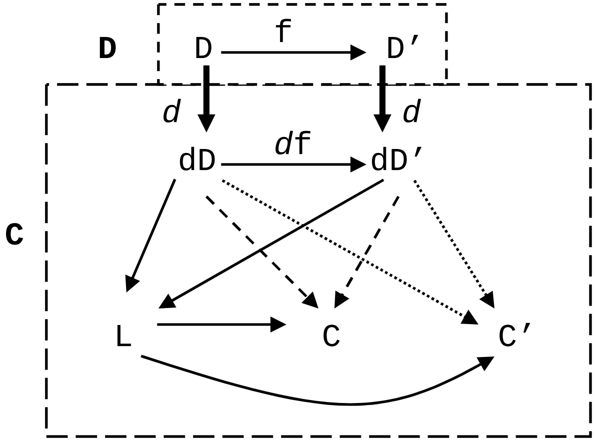
- Category **C**
  - Collection of objects |**C**|
  - Collection of arrows ||**C**|| (with *dom* and *cod*)
  - Composition of arrows
  - Identity arrow for each object
- Examples
  - Category of algebras
    - The objects are algebras
    - The arrows are homomorphisms between algebras
  - Category of sets
    - The objects are sets
    - The arrows are functions

Concrete Example

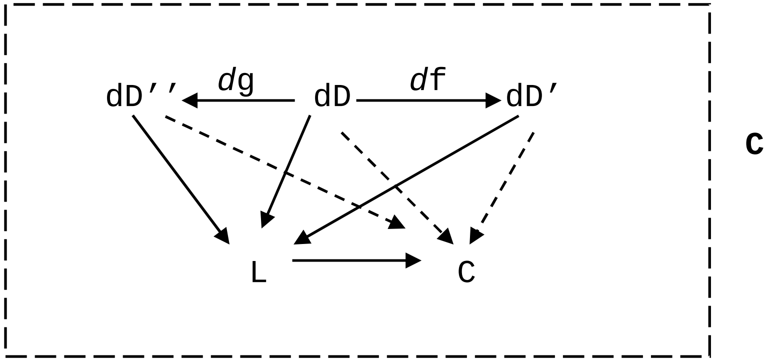
```
sort Bit;
operations
 zero:-> Bit;
 one:-> Bit;
 succ:Bit -> Bit;
equations
 succ(zero) = one;
 succ(one) = zero;
```



Colimit and Pushout



Colimit



Pushout

Functor *d*:**D** -> **C** is called a diagram of shape **D** in category **C**

Institution Theory

- Formalizes:

  Truth is invariant under changes of notation
- Institution (**Sign**,*Mod*,*Sen*, $\models$)
  - **Sign**: category of signatures
  - *Sen*: **Sign** → **Set** functor giving set of sentences for each signature
  - *Mod*: **Sign** → $\mathbf{Cat}^{op}$ functor giving category of models for each signature
  - $\models_\Sigma \subseteq |Mod(\Sigma)| \times Sen(\Sigma)$ signature-indexed family of satisfaction relations such that for

$$(\phi:\Sigma \to \Sigma') \in ||Sign||, e \in Sen(\Sigma),\ M' \in |Mod(\Sigma')|$$

$$M' \models_{\Sigma'} Sen(\phi)(e)\ \textit{if and only if}\ \ Mod(\phi)(M') \models_\Sigma e$$

MODULAR SEMANTICS

Static Semantics – Programming in the small

- Notion of fixed data
- Notion of invariance
- Signature (S_{Stc}, Σ_{Stc})
  - S_{Stc} set of sorts
  - Σ_{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* $f_s(\alpha(\sigma)(a_1, \ldots, a_n)) = \alpha'(\sigma)(f_{s_1}(a_1), \ldots, f_{s_n}(a_n))$
- 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

  $$a^{*}(t1) = a^{*}(t2) \;\; for \quad every \quad assignment \quad a : X \to |A_{Stc}|,$$

  $$A_{Stc} \models_{\Sigma_{Stc}} e$$

- Institution for static algebras (equational-[Goguen]) $(Sig_{Stc}, \mathrm{Alg}_{Stc}, Eqn_{Stc}, \models_{Stc})$
  - Sig_{Stc} category of static signatures and morphisms
  - Alg_{Stc} functor giving category of static algebras for each signature
  - Eqn_{Stc} functor giving a set of equations for each signature
  - $\models_{Stc}$ satisfaction such that

    $$A'_{Stc} \models_{\Sigma'_{Stc}} \varphi(e) \quad iff \quad A'_{Stc}|_{\varphi} \models_{\Sigma_{Stc}} e \quad with \quad \varphi : \Sigma_{Stc} \to \Sigma'_{Stc}$$

Static Semantics – Specification construction

- Specification extension
  - Extension satisfies *no confusion* and *no junk* constraint
  - $(S'_{Stc}, \Sigma'_{Stc}, E') \quad extends \quad (S_{Stc}, \Sigma_{Stc}, E) \Rightarrow S_{Stc} \subseteq S'_{Stc}, \Sigma_{Stc} \subseteq \Sigma'_{Stc}, E \subseteq E'$
  - 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

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}, Σ_{SB})
  - $S_{SB} = (State, S_V)$
  - $\Sigma_{SB} = (isInit, \Upsilon, next, \Phi, \Omega, \Delta)$
    - Υ set of generalized hidden constants $cst : S_{V_{0,\ldots,n}} \rightarrow State$
    - Φ optional set of operations $\phi : State \times S_{V_{0,\ldots,n}} \rightarrow State$
    - Ω set of attributes $\omega : State \times S_{V_{0,\ldots,n}} \rightarrow S_V$
    - Δ set of data operations $\delta : S_{V_{0,\ldots,n}} \rightarrow S_V$
    - Distinction between operators of Υ and $next$

State-based Semantics – Programming in the small

- A state-based signature: hidden signature[Goguen]
  - Hidden sort = *State*
  - Visible data universe = (S_V, Δ, D_{SB})
  - At most one hidden sort occurs in Υ or Ω
- Behavioral Satisfaction
  - A context of sort *h* is a visible sorted Σ-term that has a single occurrence of a new variable symbol *z* of sort *h*, e.g. x(z), x(next(z)).
  - A hidden algebra behaviorally satisfies equation e

$$A \mid\equiv_{\Sigma} (\forall X) t = t' \quad if \quad t_1 = t_1^{'}, \ldots, t_m = t_m^{'}$$

$$iff \quad for \quad each \quad appropriate \quad context \quad c \quad and \quad assignment \quad \theta : X \rightarrow A$$

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

$$whenever \quad \theta^*(c_j[t_j]) = \theta^*(c_j[t_j^{'}] \quad for \quad j = 1, \ldots, m \quad and \quad all \quad appropriate \quad c$$

State-based Semantics – Programming in the small

- State-based specification (S_{SB}, Σ_{SB}, E)
  - (S_{SB}, Σ_{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_{Δ} 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 Ψ-operations
    - Local term: every proper subterm is a Ψ-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,Ψ)
  - Maps hidden sorts to hidden sorts

$$\mathit{morphism} \quad (S_{SB},\Sigma_{SB}) \rightarrow (S'_{SB},\Sigma'_{SB})$$

$$\mathit{signature} \quad \mathit{morphism} \quad \varphi : \Sigma_{SB} \rightarrow \Sigma'_{SB}$$

$$\mathit{if} \quad \sigma' \in \Phi' \quad \mathit{or} \quad \sigma' \in \Omega' \quad \mathit{then} \quad \exists \quad \sigma \in \Phi \quad \mathit{or} \quad \sigma \in \Omega \,|\, \sigma' = \varphi(\sigma)$$

  - 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

$$Sign_{SB}$$

  - Functor giving a set of equations for each signature

$$Sen_{SB}$$

  - Functor giving a category of hidden algebras for each signature

$$Mod_{SB}$$

  - Satisfaction relation

$$|\!\equiv_{\Sigma_{SB}}$$

  - Satisfaction condition

$$A'|_{\varphi} |\!\equiv_{\Sigma_{SB}} e \quad iff \quad A' |\!\equiv_{\Sigma'_{SB}} \varphi(e)$$

State-based Semantics – Coalgebras

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

$$(Set^{S_{SB}}_{D_{SB}}, F_{\Sigma_{SB}}) \quad with \quad F_{\Sigma_{SB}} : Set^{S_{SB}}_{D_{SB}} \to Set^{S_{SB}}_{D_{SB}}$$

$$(X_{S_1},\dots,X_{S_n},X_{State}) \to (X_{S_1},\dots,X_{S_n},\prod_{k\in 1,\dots,l} X_{S_k}^{X_{S_{0,\dots,n}}} \times X_{State}^{X_{S_{0,\dots,n}}})$$

- Example:
  - State-based signature $State, Natural$

    $s_0 :\to State, x : State \to Natural, next : State \to State, \Delta_{Natural}$
  - Destructor hidden subsignature

    $(\{Natural, State\}, \{x : State \to Natural, next : State \to State\} \cup \Delta_{Natural})$
  - Associated abstract cosignature

    $(Set_N^{\{Natural, State\}}, F) \quad with \quad FX_{State} = N \times X_{State}$
  - A coalgebraic structure

    $\alpha : X_{State} \to N \times X_{State}$

State-based Semantics – Specification construction

- Extension: similar in essence to static specification extension
  - The signature morphism is reverse

$$(S_{SB},\Sigma_{SB},E)\ {}^{c}\!\rightarrow (S'_{SB},\Sigma'_{SB},E') \quad iff \quad \exists \varphi : (S'_{SB},\Sigma'_{SB}) \rightarrow (S_{SB},\Sigma_{SB})$$

- 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

$$\begin{tikzcd}
 & \text{SpecShared} \arrow[dl] \arrow[dr] \arrow[dd] & & \\
\text{Spec1} \arrow[dr] & & \text{Spec2} \arrow[dl] & \text{OtherSpec2} \arrow[l, dashed] \\
 & \text{Spec3 = Spec1 + Spec2} & &
\end{tikzcd}$$

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

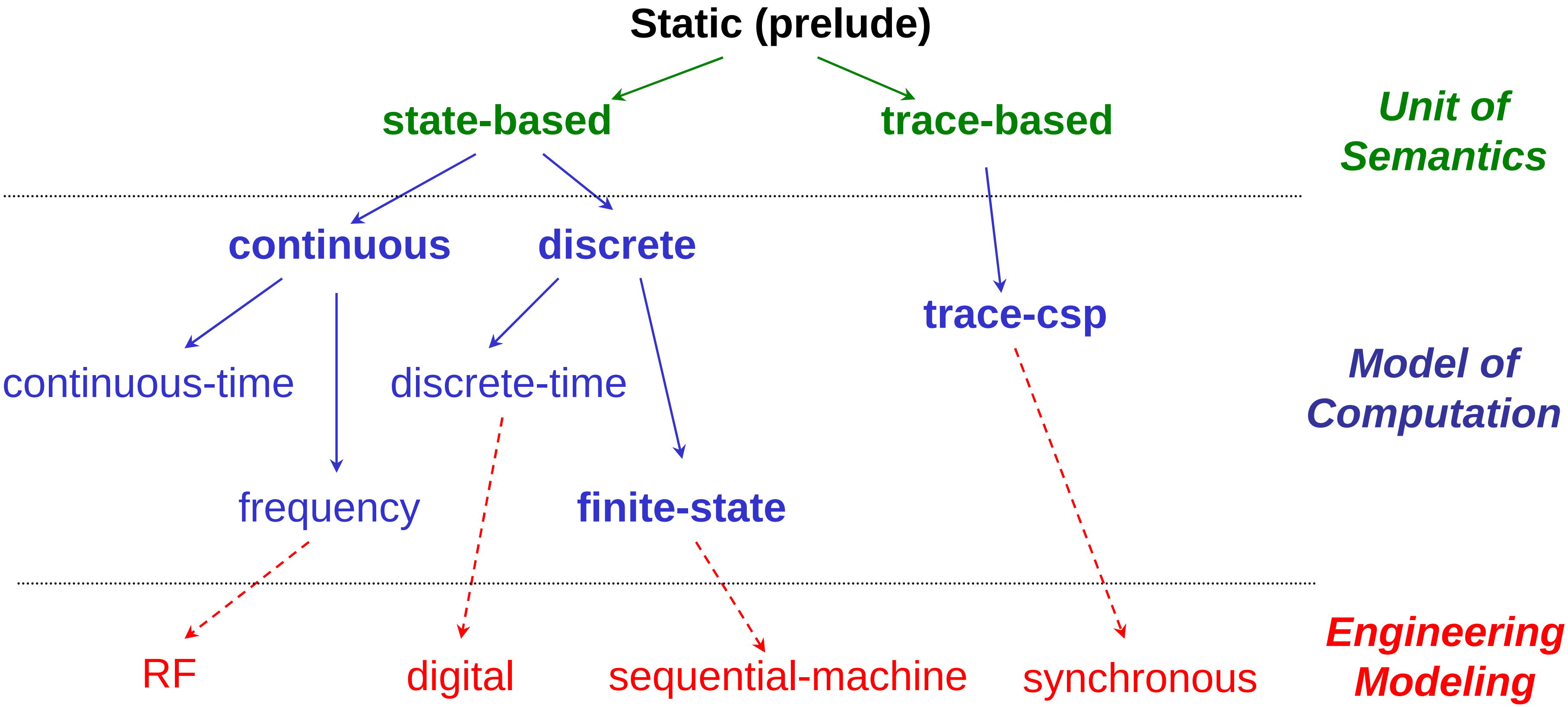
Specification Translation from State-based to Trace-based

- 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 $T_{St} :: Trace(State)$ 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

$$\begin{array}{l} someTrace \in T_{St}, s \in State \quad such \quad that \quad someTrace[n] = s \\ and \quad next(s, I_0[n], \ldots, I_k[n]) = someTrace[n+1] \end{array}$$

SPECIFICATION IN ROSETTA

The Domain organization



Static (prelude)

state-based

trace-based

Unit of Semantics

continuous

discrete

trace-csp

continuous-time

discrete-time

frequency

finite-state

Model of Computation

RF

digital

sequential-machine

synchronous

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)

$$S_{Stc} = \{\ldots, Boolean, \ldots\}$$

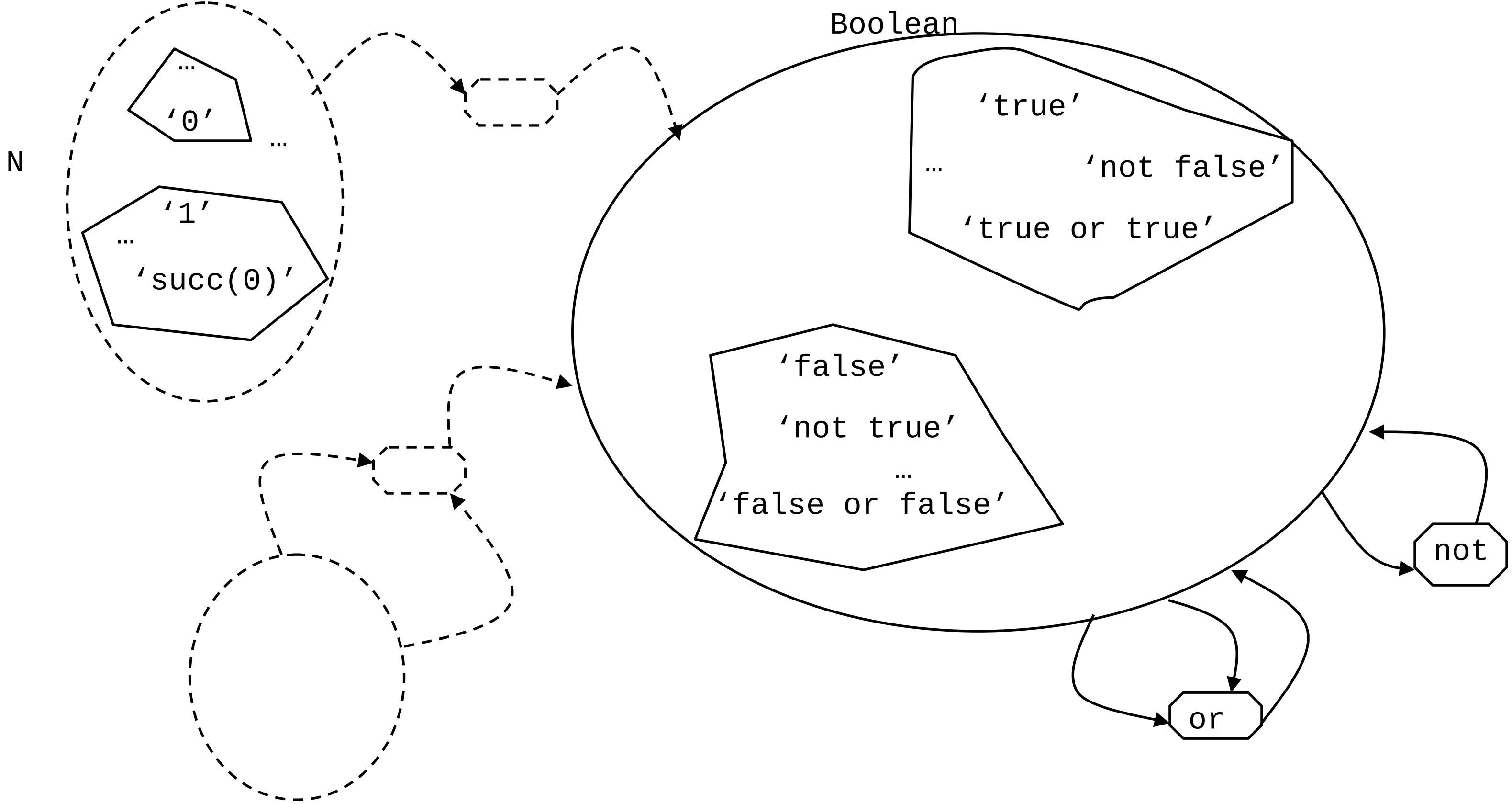
$$\Sigma_{Stc} = \{\ldots, false :\to Boolean, true :\to Boolean, not : Boolean \to Boolean, \ldots$$

$$\ldots, or : Boolean \times Boolean \to Boolean, \ldots\}$$

Static Domain Specification

```
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



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

$$S_{SB} = (State, S_{Stc})$$

$$\Sigma_{SB} = (isInit, \Upsilon_{SB}, next, \{\}, \{\}, \{\_\_@\_\_\} \cup \Sigma_{Stc})$$

- Coalgebras

$$|A|_{State} \xrightarrow{\gamma_{next}} \{*\} \cup |A|_{State}$$

$$|A|^{R}_{State} \xrightarrow{\zeta} |A|^{R}_{State}$$

State-based Domain Specification

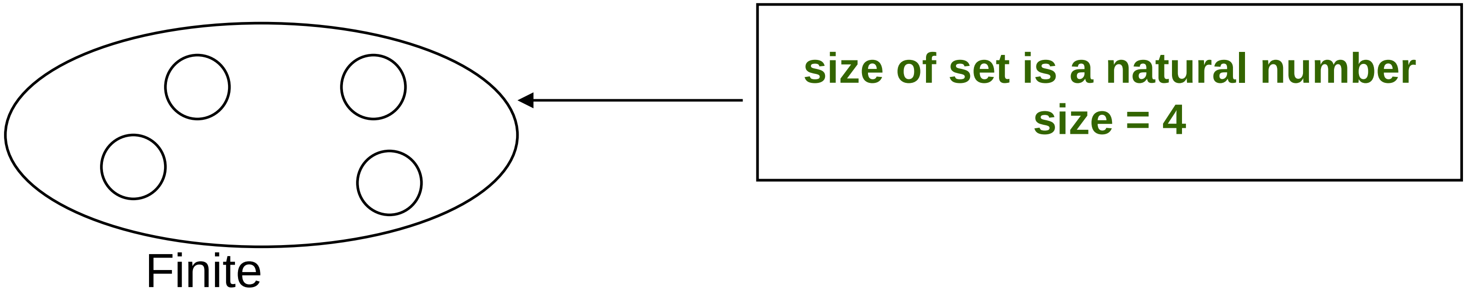
```
domain state_based(State::design Type) :: static is

  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

```
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;
```

The Finite-state Domain

Finite-state ⇒ observations are finite and discrete



size of set is a natural number
size = 4

```
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 ⇒ all observations have continuous variations with respect to a continuous observation of states

$$\boxed{\frac{\Delta f}{\Delta s} = \frac{f(next(s)) - f(s)}{contAttr(next(s)) - contAttr(s)}}$$

```
domain continuous :: state_based is
 contAttr(st::State)::Real;
 variation[T::Type](fnc::<*stt::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

Trace-based Domain Specification

```
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

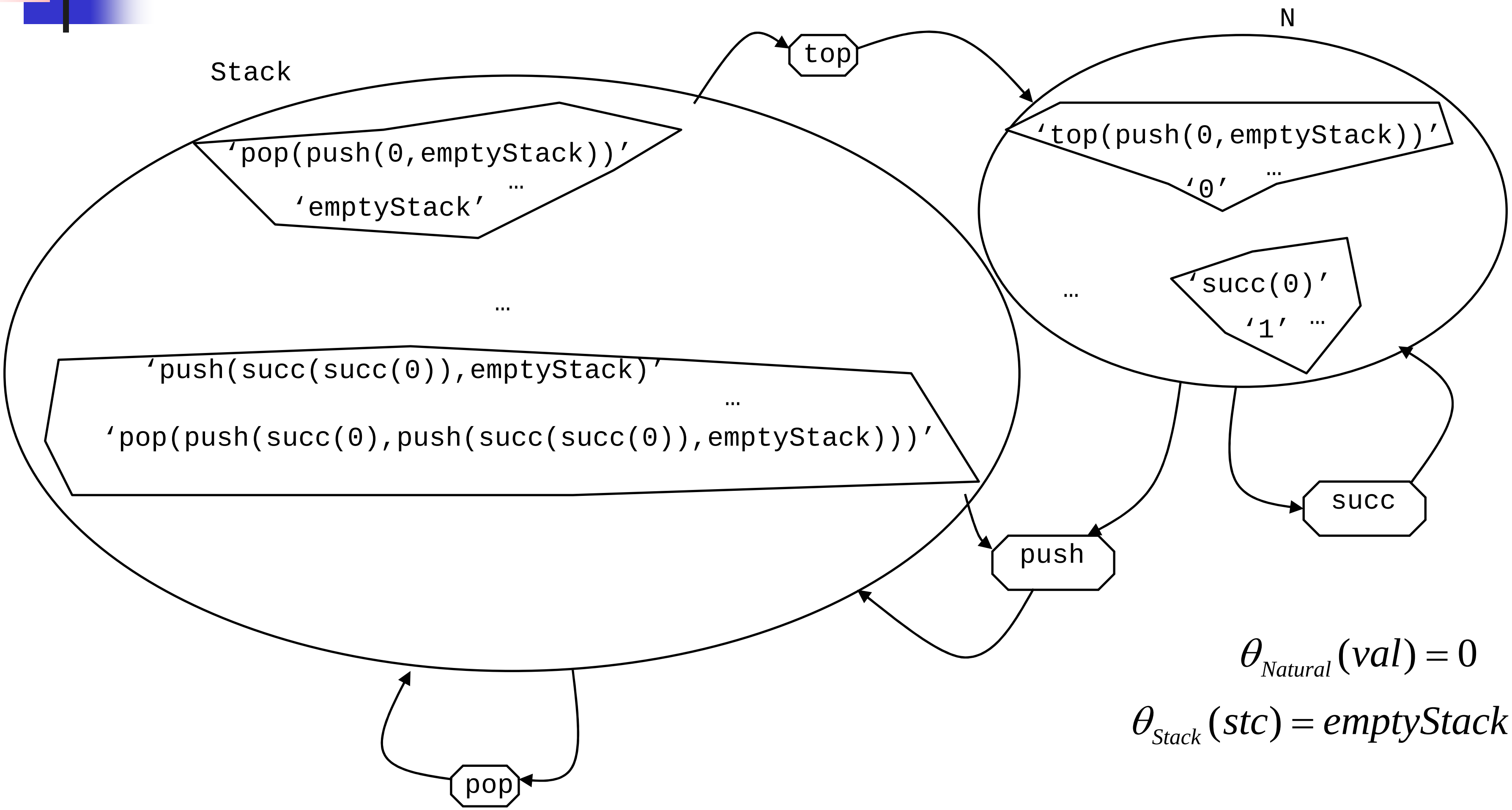
```
facet stackDT::static is
  Stack::type;
  emptyStack::Stack is constant;
  push(stcParam::Stack; n::Natural)::Stack;
  pop(stcParam::Stack)::Stack;
  top(stcParam::Stack)::Natural;
  val::Natural;
  stcVar::Stack;
begin
  pop_empty: pop(emptyStack) = emptyStack;
  top_empty: top(emptyStack) = 0;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

$$S_{stackDT} = S_{Stc} \cup \{Stack\}$$

$$\Sigma_{stackDT} = \Sigma_{Stc} \cup \{emptyStack, push, pop, top\}$$

$$E_{stackDT} = E_{Stc} \cup \{pop\_empty, top\_empty, pop\_push, top\_push\}$$

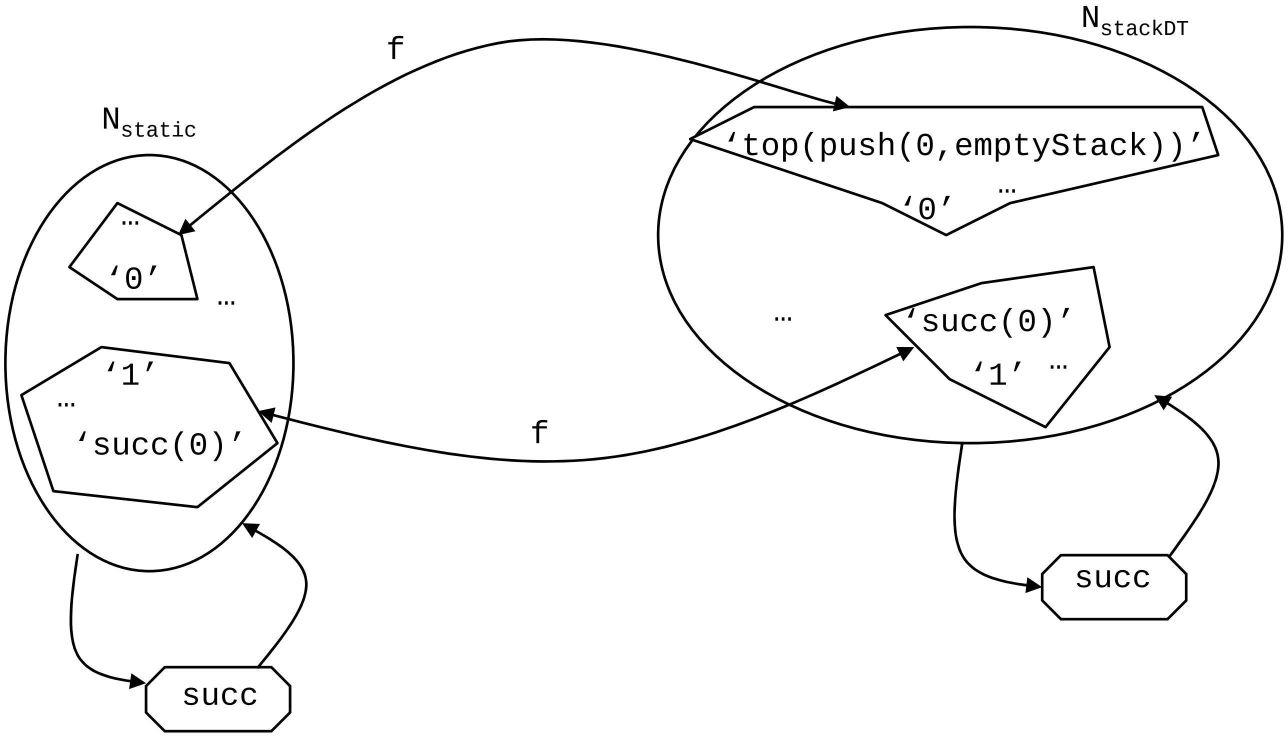
Initial algebra for stackDT



$$\theta_{Natural}(val) = 0$$

$$\theta_{Stack}(stc) = emptyStack$$

Isomorphism between $N_{stackDT}$ and N_{Stc}



$N_{stackDT}|_\varphi = N_{stackDT}$

$N_{stackDT}|_\varphi$ satisfies *static* and is isomorphic to N_{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

$$\begin{tikzcd}
 & \begin{array}{c}\texttt{Discrete(StateSet)}\\ \texttt{next::Function}\end{array} & \\
\begin{array}{c}\texttt{memoryA(val)}\\ \texttt{memA::State} \to \texttt{Natural}\\ \texttt{next(st::State;val::Natural)::State}\end{array} \arrow[ur] & & \begin{array}{c}\texttt{memoryB(val)}\\ \texttt{memB::State} \to \texttt{Natural}\\ \texttt{next(st::State;val::Natural)::State}\end{array} \arrow[ul] \\
 & \begin{array}{c}\texttt{twoMemory(val)}\\ \texttt{memA::State} \to \texttt{Natural}\\ \texttt{memB::State} \to \texttt{Natural}\\ \texttt{next(st::State;val::Natural)::State}\end{array} \arrow[ul] \arrow[ur] \arrow[uu, dashed] &
\end{tikzcd}$$

Pullback of Signature Morphisms

Composition of Parameterized Specifications

$$\begin{tikzcd}
 & {|A|_{State}} \arrow[d, "{\gamma_{discrete} = next}"] & \\
 & {|A|^{N}_{State}} \arrow[dl] \arrow[dr] & \\
{|A|_{State}} \arrow[d, "{\gamma_{memoryA} = (memA, next)}"] & & {|A|_{State}} \arrow[d, "{\gamma_{memoryB} = (memB, next)}"] \\
{N \times |A|^{N}_{State}} \arrow[dr] & & {N \times |A|^{N}_{State}} \arrow[dl] \\
 & {|A|_{State}} \arrow[d, "{\gamma_{twoMemory} = (memA, memB, next)}"] & \\
 & {N \times N \times |A|^{N}_{State}} &
\end{tikzcd}$$

Pushout of Coalgebras

Trace-based MemoryA Specification

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

facet traceMemA(val::input Trace(Natural))::trace_based() is
  memA(st::State)::Natural;
  StateTrace::Trace(State);
  someTrace::StateTrace;
  s::State; next::Function; … // All declarations from domains
  pos::Natural;
 begin
   initA: isInit(s) => ((memA(s) = 0) and (pos = 0));
   next_def: next = memNext;
   lA: memA(next(s,getEventAt(val,pos))) = getEventAt(val, pos);
   newT1: getEventAt(someTrace,pos) = s;
   newT2: next(s,getEventAt(val,pos)) = getEventAt(someTrace, pos+1);
end facet traceMemA;
```

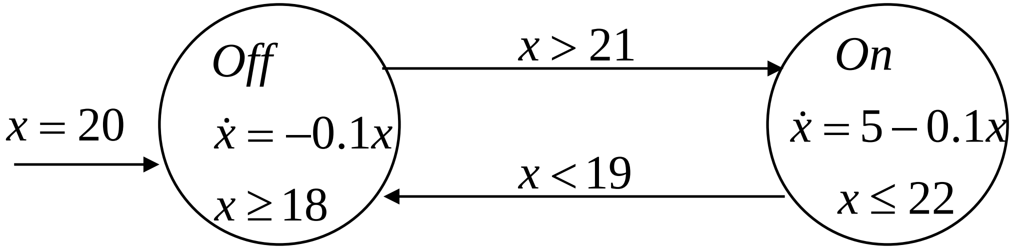
$$|A|_{Trace(State)} \xrightarrow{\gamma_{traceMemA}} N \times (|A|_{Trace(State)})$$

$$\gamma_{traceMemA} = (memA(head), tail)$$

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)

Hybrid Automaton of a Thermostat



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

```
facet heater(x::input Real; ctrl::output ControlMode):: finite_state is
 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;
```

The Temperature Specification

```
facet 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;
```

The Thermostat Specification

```
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 *TemperatureVariation* specifications
- Models that satisfy *Thermostat* will have (minimal) states as pairs (*controlmode*,*temp*) with *controlmode*=ctrl(s) and *temp*=x(s)
- *Controlmode*: on or off
- *Temp*: a real number between 18 and 22
- If considering discrete *Thermostat* models, *temp* will have discretized values through “sampling”

RELATED WORK
AND
FUTURE WORK

Related Work

- CafeOBJ – http://www.ldl.jaist.ac.jp/cafeobj
- Ptolemy II – *Heterogeneous Concurrent Modeling and Design in Java* – J. Davis, C. Hylands, B. Kienhuis, E. Lee, et al.; University of California at Berkeley
- Metropolis – *Overcoming Heterophobia: Modeling Concurrency in Heterogeneous Systems* – J. Burch, R. Passerone, A. Sangiovanni-Vincentelli

Related Work

- SAL – *An Overview of SAL* – J. Rushby, S. Owre, N. Shankar, A. Tiwari et al.
- Viewpoints Modeling – *Viewpoints: A Framework for Integrating Multiple Perspectives in System Development* – A. Finkelstein et al.

- 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

- The MultiGraph Architecture – *Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments* – G. Nordstrom et al.; Vanderbilt University
- GME – *The Generic Modeling Environment* – A. Ledeczi et al.; Vanderbilt University
- UML-Metamodeling Architecture – *An UML-metamodeling Architecture for Interoperability of Information Systems* – M. Terrasse et al.

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