Polymorphic Scenario-Based Specification Models: Semantics and Applications

Shahar Maoz
The Weizmann Institute of Science, Israel
Presentation at MoDELS 2009, Scientific Track

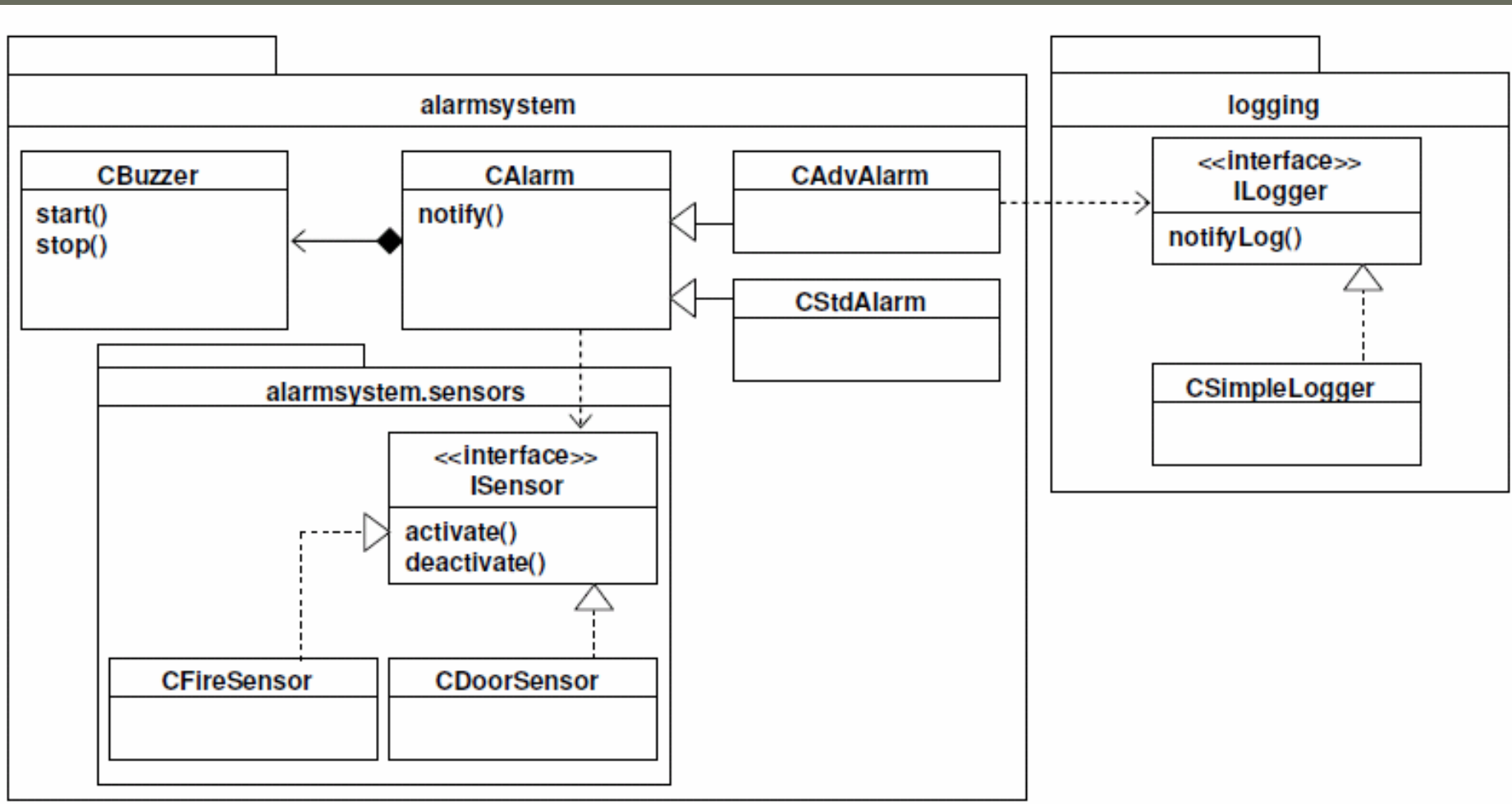
One Slide Abstract

- Polymorphic scenario-based specifications
  - a generalization of UML2 interactions in the context of object-orientation
  - interaction lifelines may represent classes and interfaces rather than concrete objects
- A polymorphic semantics for interactions
  - formal trace-based semantics, using automata extended with an ad-hoc binding mechanism
- Inter-object behavior common to all objects of a certain type can be specified at the most abstract level where it is applicable, producing succinct and reusable specifications
- The work is done in the context of live sequence charts [DH01], which extend classical sequence diagrams with existential/universal cold/hot modalities; allows to specify liveness and safety properties
- Applications: specification, testing, execution

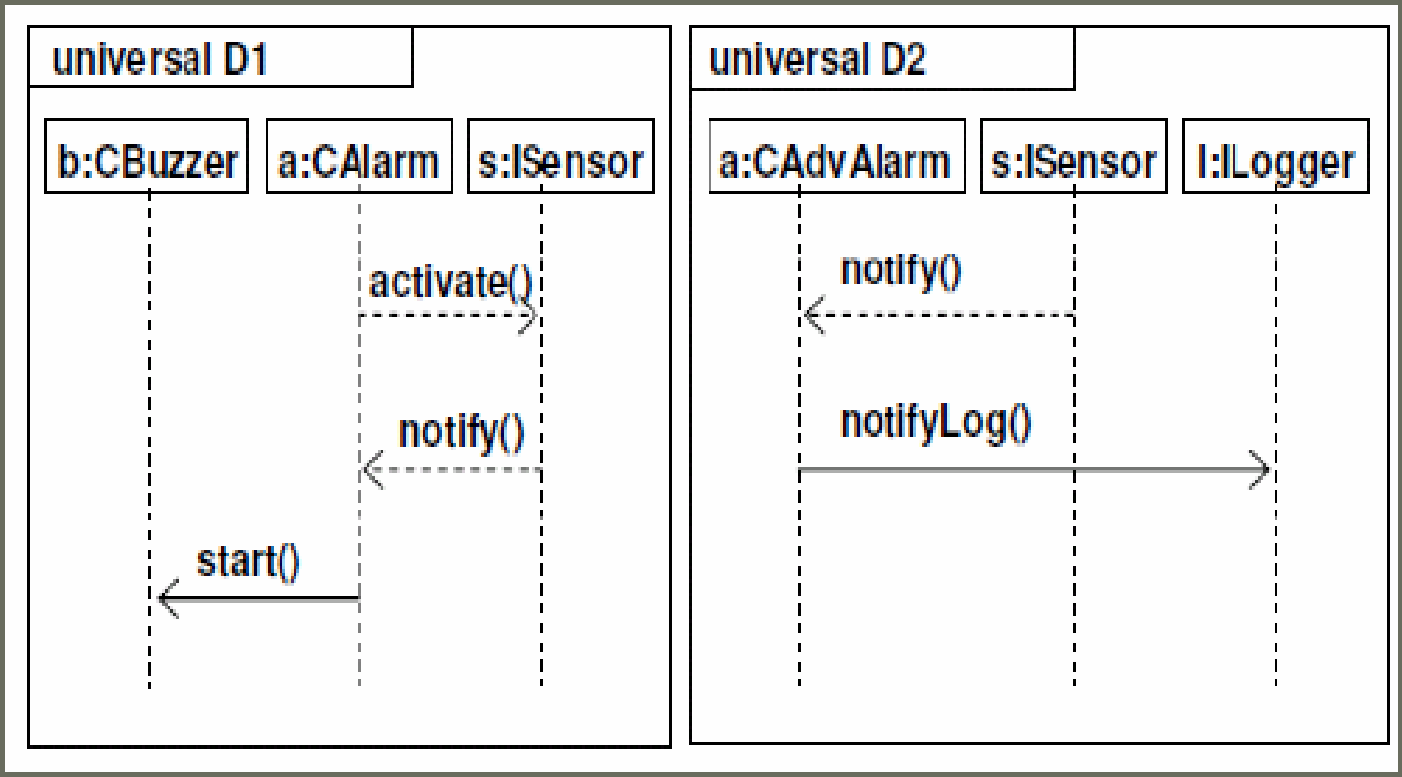
Polymorphism

- The ability of a type T1 to appear and be used like another type T2
- A fundamental characteristics of object-oriented design, enabling important features such as modularity and reuse
- We take advantage of polymorphism in the context of scenario-based specifications

Example: Alarm System



Example: Alarm System



tr1: <cstdalarm,activate,fs1><cstdalarm,activate,ds1><fs1,notify,cstdalarm>
<cstdalarm,start,cbuzzer><cstdalarm,activate,ds1>...

tr2: <cadvalarm,activate,fs3><cadvalarm,activate,ds1><fs3,notify,cadvalarm>
<cadvalarm,notifyLog,simplelogger><cadvalarm,start,cbuzzer>...

Key Idea

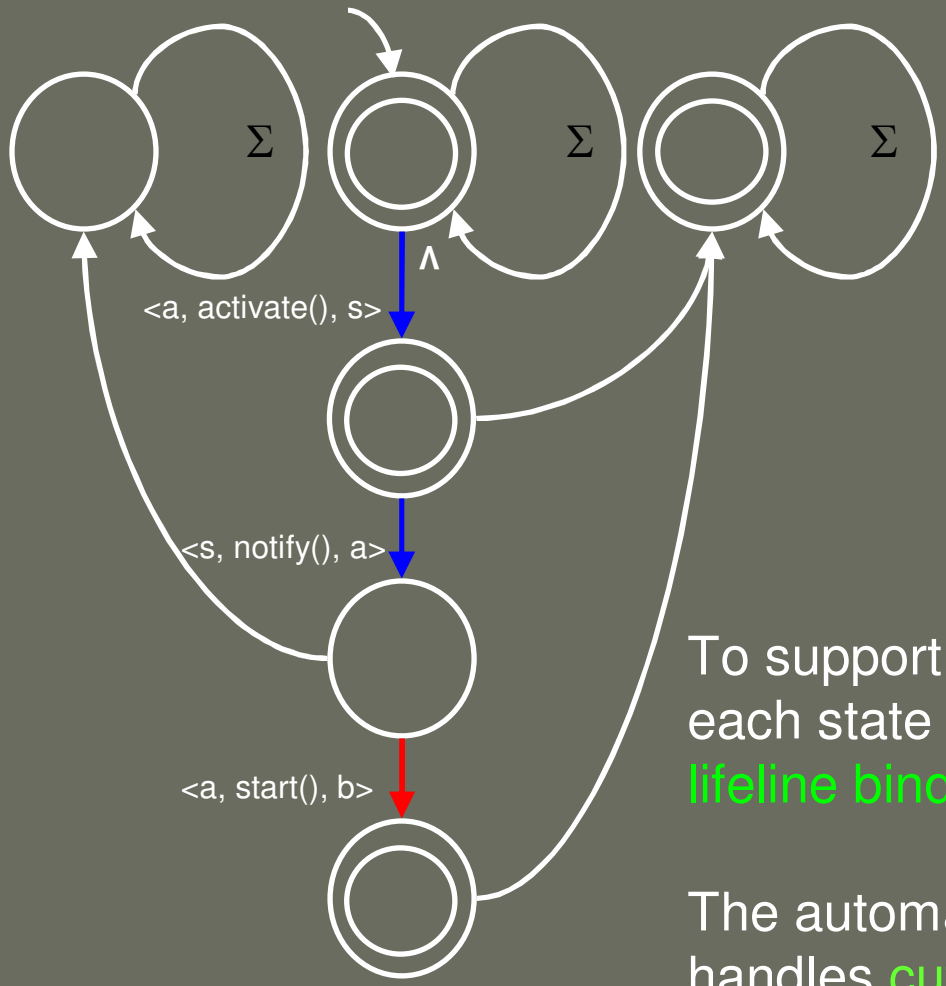
Inter-object behavior common to all objects derived directly or indirectly from a certain type, can be formally specified at the most abstract level where it is applicable, instead of being repeated for each class (or worse, for each object)

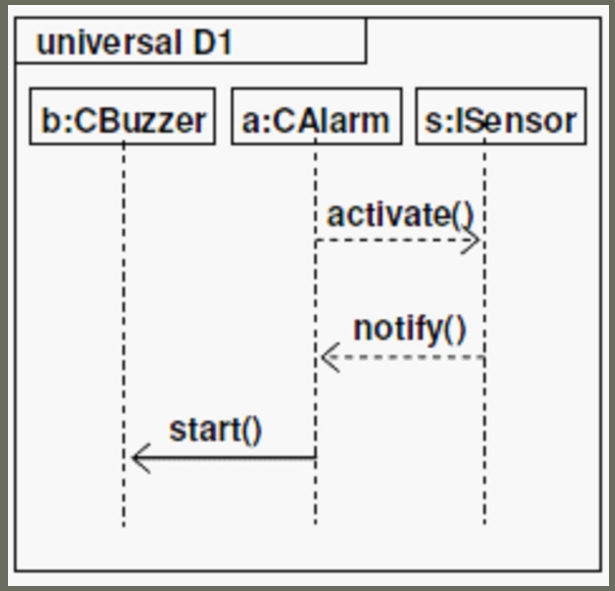
In essence, an inter-object application of behavioral subtyping; subtyping modulo user-defined interactions

Semantics

- The semantics of a scenario is given using an automaton
  - alphabet <caller, signature, callee>
  - states represent cuts along the progress of the scenario
  - transitions are labeled with enabled / violating events
  - accepting states refer to cold (stable) cuts
- The trace-language accepted by the automaton is the language defined by the scenario
- We build on the semantics of Damm and Harel's live sequence charts (LSC) [DH01], and adopt the modal profile for UML2 interactions from [HM08]

Automaton Construction (Simplified)





To support polymorphism, we add to each state of the automaton a tuple of lifeline binding types

The automaton's transition function handles cut changes and bindings

Additional technical details in the paper…

More formally…

System-model We consider a *system-model* $Sys = \langle O, Ty, type, \leq_{Ty} \rangle$, which includes a (possibly infinite) set of objects $O = \{o_1, o_2, \ldots\}$, a partially ordered set of types $Ty = \{ty_1, ty_2, \ldots, ty_m\}$, and a mapping from each object in O to its type $type : O \longrightarrow Ty$. The mapping $type$ derives an $instanceof$ Boolean function $instanceof : (O \times Ty) \longrightarrow \{true, false\}$ such that $instanceof(o, ty) = true$ iff $type(o) \leq_{Ty} ty$.

A type $ty \in Ty$ has a finite set of method signatures $m(ty) = \{m_1, m_2, \ldots, m_s\}$. The subtyping partial order $\leq_{Ty}$ over Ty implies signatures set inclusion: $\forall ty_1, ty_2 \in Ty, ty_1 \leq_{Ty} ty_2$ implies $m(ty_2) \subseteq m(ty_1)$. We allow multiple inheritance with a disjoint signatures restriction: $\forall ty_1, ty_2, ty_3 \in Ty$, if $ty_1 \leq_{Ty} ty_2$ and $ty_1 \leq_{Ty} ty_3$ and $ty_2 \not\leq_{Ty} ty_3$ and $ty_3 \not\leq_{Ty} ty_2$ then $m(ty_2) \cap m(ty_3) = \emptyset$. Note that we ignore the difference between class and interface types as it has no semantic significance in the trace-based semantics we present.

A *system-model event* e is a tuple $\langle o_{src}, m, o_{trg} \rangle$ where $o_{src}, o_{trg} \in O$ and $m \in m(type(o_{trg}))$, carrying the intuitive meaning of object o_{src} calling method m of object o_{trg} (we allow $o_{src} = o_{trg}$). A *system-model trace* is an infinite sequence of events $e_1, e_2, e_3, \ldots$.

More formally…

The set of lifelines L and the mapping $ltype$ define the set of possible bindings $Bind(L) \subseteq (O \cup \{\perp\})^k$ such that $\langle o_1, o_2, \ldots, o_k \rangle \in Bind(L)$ iff $\forall i, 1 \leq i \leq k, o_i = \perp \vee instanceof(o_i, ltype(l_i))$. A given binding $\langle o_1, o_2, \ldots, o_k \rangle \in Bind(L)$ defines a trivial projected function $bind : L \longrightarrow (O \cup \{\perp\})$ from a lifeline to its bound object: $\forall i, 1 \leq i \leq k, bind(l_i) = o_i$.

Given a universal diagram D we construct an alternating Büchi automaton $A_D = \langle \Sigma \cup \epsilon, Q, q_{in}, \delta, \alpha \rangle$, where

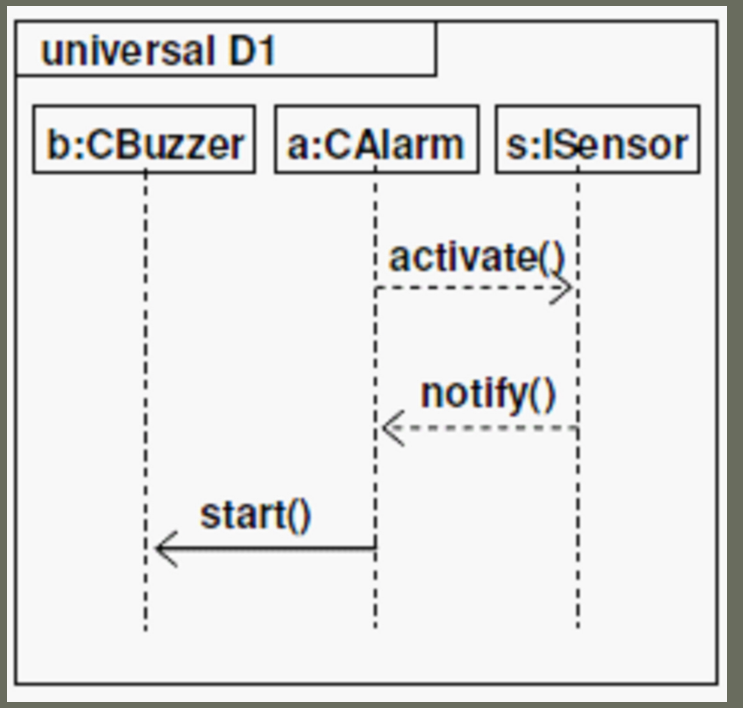
- $\Sigma = \{\langle o_1, m, o_2 \rangle | o_1, o_2 \in O \wedge m \in m(type(o_2))\}$;
- $Q = S \times Bind(L) \cup \{q_{rej}, q_{acc}\}$ is a set of states (we use $cut(q)$ to denote the cut-state s of a state $q = \langle s, \langle o_1, \ldots, o_k \rangle \rangle$);
- $q_{in} = \langle s_{min}, \langle \{\perp\}^k \rangle \rangle$ is the initial state;
- $\alpha = \{\langle s, \langle o_1, \ldots, o_k \rangle \rangle | mode(s) = cold\} \cup \{q_{acc}\}$ is the accepting condition (that is, all cold states and q_{acc} are accepting);
- and $\delta : Q \times \Sigma \longrightarrow B^+(Q)$ is a transition function defined as follows:

More formally…

- (otherwise, the source object of cme is already bound and the target can bind to a free lifeline)
  for l^i_{src} s.t. $source(cme) = bind(l_{src})$
  for all l^j_{trg} s.t. $instanceof(target(cme), ltype(l_{trg})) \wedge bind(l_{trg}) = \perp \wedge \langle l^i_{src}, m(cme), l^j_{trg} \rangle \in EnLPME(cut(q))$:
  $\delta(q, cme) = \bigwedge_{l^j_{trg}} \langle R(cut(q), e), \langle \overline{o}_1, \ldots, \overline{o}_{l_{trg}}, \ldots, \overline{o}_k \rangle \rangle$
  where $e = \langle l^i_{src}, m(cme), l^j_{trg} \rangle \wedge \overline{o}_{l_{trg}} = target(cme) \wedge \forall h \neq l_{trg} : \overline{o}_h = o_h$;

Semantic Issues

- Multiple copies
- Multiple binding choices
- Combining static and dynamic binding
- Single binding constraint
- Some more in the paper…



Application: Testing

- Polymorphic scenarios can be used to specify polymorphic inter-object tests
- We use the S2A compiler [MH06, HKM07] to transform UML2 sequence diagrams extended with the modal profile into monitoring scenario aspects
- Each generated scenario aspect simulates an automaton whose states correspond to scenario cuts; pointcuts listen out for relevant system events; advice advances the automaton state

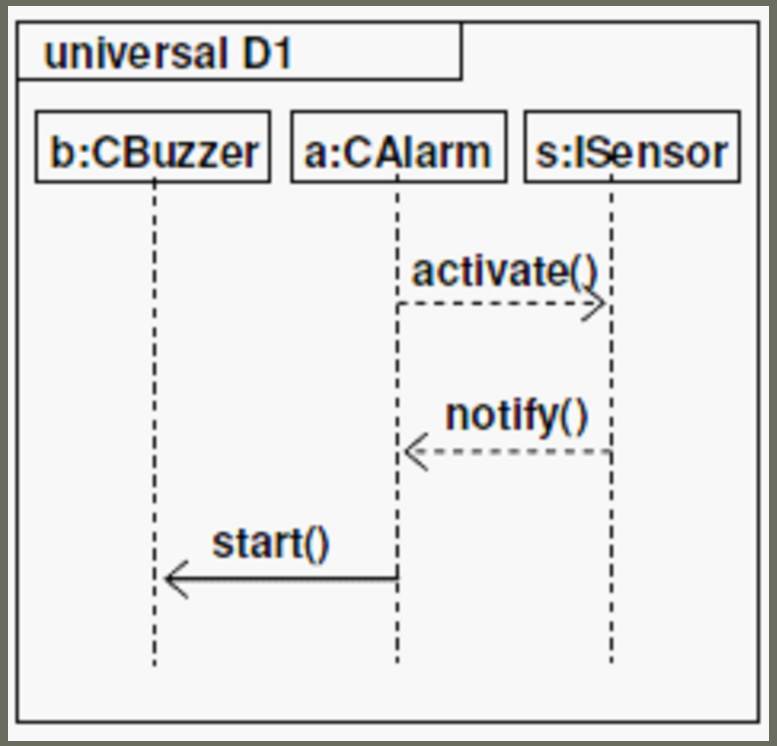
Application: Testing (cont.)

- To handle polymorphic scenarios, S2A takes advantage of Java and AspectJ support for inheritance and interface implementation
- The activation and progress of the scenarios are monitored by the generated aspects
- Completions and violations are reported using a scenario-based trace [Maoz08] and can be visualized and interactively analyzed using the Tracer [MKH07]

Application: Testing (cont.)

- The generated aspect code monitors the progress of the scenarios, waits for the hot events to happen, checks for violations

- Polymorphic scenarios enable
  - succinct test specification
  - reusable test specification



Application: Execution

- S2A supports the execution of polymorphic scenarios, following the play-out algorithm [Harel and Marelly 2003]
- Some interesting semantic issues arise:
  - executing ‘abstract’ enabled methods
  - another non-determinism dimension
- Current implementation in S2A is naïve
  - requires further definition and implementation work

Related Work

- Interaction Pattern Specifications [FKGS04]
  - lifelines are labeled with role names
  - conformance rules are defined between a pattern and its concretization
  - polymorphism not considered
- LSCs with symbolic lifelines [MHK02]
  - Lifelines may represent types
    - but inheritance and interface realization not defined
  - Play-out execution implemented in the Play-Engine tool [HM03]
  - Operational semantics defined
    - but trace-based semantics not defined
- Several different efforts towards a semantics for UML2 interactions; it seems none considers the relationship between interactions and a polymorphic object-oriented system-model

Summary of Contribution

- A generalization of UML2 interactions in the context of object-oriented system models
- Powerful language, enables succinct, reusable inter-object specifications
  - formal trace-based semantics
  - some interesting semantic issues
- Applications

Discussion

- Subtyping
  - structural intra-object subtyping
  - behavioral intra-object subtyping
  - behavioral subtyping modulo inter-object scenarios
- Is the behavioral subtyping supported by polymorphic scenarios ‘too powerful’?
  - do we need an overriding mechanism between interactions?

Thank You!