Transition Systems and Service Composition

Giuseppe De Giacomo

Seminari di Ingegneria del Software
A.A. 2006/2007

Transition Systems
Concentrating on behaviors: *SUM two integers*

- Consider a program for computing the sum of two integers.
- Such a program has essentially two states
  - the state $S_0$ of the memory before the computation: including the two number to sum
  - the state $S_1$ of the memory after the computation: including the result of the computation
- Only one action, i.e. “sum”, can be performed

```
        sum
S0 ----> S1
```

Concentrating on behaviors: *CheckValidity*

- Consider a program for computing the validity of a FOL formula:
- Also such a program has essentially two states
  - the state $S_1$ of the memory before the computation: including the formula to be checked
  - the state $S_2$ of the memory after the computation: including “yes”, “no”, “time-out”
- Only one action, i.e. “checkValidity”, can be performed

```
        checkValidity
S0 ----> S1
```
Concentrating on behaviors

- The programs SUM and CheckValidity are very different from a computational point of view.
  - SUM is trivial
  - CheckValidity is a theorem prover hence very complex
- However they are equally trivial from a behavioral point of view:
  - two states $S_1$ and $S_2$
  - a single action $\alpha$ causing the transition

\[
\begin{align*}
  S_0 & \xrightarrow{\alpha} S_1
\end{align*}
\]

Concentrating on behaviors: RockPaperScissor

- Consider the program RockPaperScissor that allows to play two players the the well-known game.
- The behavior of this program is not trivial:
Concentrating on behaviors: 
*RockPaperScissor* (automatic)

- Consider a variant of the program RockPaperScissor that allows one players to play against the computer.
- The behavior of this program is now **nondeterministic**:

```
  1chooses
  /   \
  |     |
  v     v
  1choices
  /   \
  |     |
  v     v
  1choices
```

Concentrating on behaviors: 
*WebPage*

http://www.informatik.uni-trier.de/~ley/db/

A web page can have a complex behavior!
Concentrating on behaviors: Vending Machine

Concentrating on behaviors: Another Vending Machine
Concentrating on behaviors: Vending Machine with Tilt

Transition Systems

- A transition system TS is a tuple $T = < A, S, S^0, \delta, F >$ where:
  - $A$ is the set of actions
  - $S$ is the set of states
  - $S^0 \subseteq S$ is the set of initial states
  - $\delta \subseteq S \times A \times S$ is the transition relation
  - $F \subseteq S$ is the set of final states

- Variants:
  - No initial states
  - Single initial state
  - Deterministic actions
  - States labeled by propositions other than Final/$\neg$Final

(c.f. Kripke Structure)
**Process Algebras are Formalisms for Describing TS**

- **Trans (a la CCS)**
  - Ven = 20c.Ven_b + 10c.Ven_s
  - Ven_b = big.collect_b.Ven
  - Ven_s = small.collect_s.Ven
- **Final**
  - √ Ven

- **TS may have infinite states** - e.g., this happens when generated by process algebras involving iterated concurrency

- **However we have good formal tools to deal only with finite states TS**

---

**Example (Clock)**

TS may describe (legal) nonterminating processes
Example (Slot Machine)

Nondeterministic transitions express choice that is not under the control of clients

Example (Vending Machine - Variant 1)
Example
(Vending Machine - Variant 2)

Inductive vs Coinductive Definitions:
Reachability, Bisimilarity, ...
Reachability

- A binary relation $R$ is a **reachability-like relation** iff:
  - $(s,s) \in R$
  - if $\exists a. s \rightarrow a s' \land (s',s'') \in R$ then $(s,s'') \in R$

- A state $s_0$ of transition system $S$ is **reachable-from** a state $s_f$ iff for all a **reachability-like relations** $R$ we have $(s_0, s_f) \in R$.

- Notably that
  - **reachable-from** is a reachability-like relation itself
  - **reachable-from** is the **smallest** reachability-like relation

*Note it is an inductive definition!*

---

Computing Reachability on Finite Transition Systems

**Algorithm** ComputingReachability

**Input:** transition system $TS$

**Output:** the **reachable-from** relation (the smallest reachability-like relation)

**Body**

```plaintext
R = \emptyset
R' = \{(s,s) | s \in S\}
while (R \neq R') {
    R := R'
    R' := R' \cup \{(s,s'') | \exists s',a. s \rightarrow a s' \land (s',s'') \in R\}
}
return R'
```

YdoB
Bisimulation

- A binary relation $R$ is a **bisimulation** iff:

  $$(s, t) \in R \text{ implies that }$$
  
  - $s$ is final iff $t$ is final
  
  - for all actions $a$
    
    - if $s \rightarrow_a s'$ then $\exists t' \cdot t \rightarrow_a t'$ and $(s', t') \in R$
    
    - if $t \rightarrow_a t'$ then $\exists s' \cdot s \rightarrow_a s'$ and $(s', t') \in R$

- A state $s_0$ of transition system $S$ is **bisimilar**, or simply equivalent, to a state $t_0$ of transition system $T$ iff there **exists** a bisimulation between the initial states $s_0$ and $t_0$.

- Notably
  
  - **bisimilarity** is a bisimulation
  
  - **bisimilarity** is the largest bisimulation

*Note it is a co-inductive definition!*

Computing Bisimilarity on Finite Transition Systems

**Algorithm** ComputingBisimulation

**Input:** transition system $T S_S = < A, S, S^0, \delta_S, F_S >$ and transition system $T S_T = < A, T, T^0, \delta_T, F_T >$

**Output:** the **bisimilarity** relation (the largest bisimulation)

**Body**

- $R = \emptyset$
- $R' = S \times T - \{(s, t) \mid s \in F_S \equiv t \in F_T\}$
- while $(R \neq R')$
  
  - $R := R'$
  
  - $R' := R' - \{(s, t) \mid \exists s', a . s \rightarrow_a s' \land \exists t' . t \rightarrow_a t' \land (s', t') \in R'\}$
    
  - $\{(s, t) \mid \exists t', a . t \rightarrow_a t' \land \exists s' . s \rightarrow_a s' \land (s', t') \in R'\}$

- return $R'$

Ydob
Example of Bisimulation

Example of Bisimulation
**Automata vs. Transition Systems**

- **Automata**
  - define sets of runs (or traces or strings): (finite) length sequences of actions
- **TSs**
  - ... but I can be interested also in the alternatives “encountered” during runs, as they represent client’s “choice points”

As automata they recognize the same language: $abc^* + ade^*$

Different as TSs

**Logics of Programs**
Logics of Programs

- Are modal logics that allow to describe properties of transition systems

- Examples:
  - HennessyMilner Logic
  - Propositional Dynamic Logics
  - Modal (Propositional) Mu-calculus

- Perfectly suited for describing transition systems: they can tell apart transition systems modulo bisimulation

HennessyMilner Logic

- \( \Phi := P \mid \neg \Phi \mid \Phi_1 \land \Phi_2 \mid \Phi_1 \lor \Phi_2 \mid [a]\Phi \mid <a>\Phi \)  
  
  \( (atomic \ proportions) \)
  \( (closed \ under \ boolean \ operators) \)
  \( (modal \ operators) \)

- Propositions are used to denote final states

- \( <a>\Phi \) means there exists an a-transition that leads to a state where \( \Phi \) holds; i.e., expresses the capability of executing action a bringing about \( \Phi \)

- \( [a]\Phi \) means that all a-transitions lead to states where \( \Phi \) holds; i.e., express that executing action a brings about \( \Phi \)
**Logics of Programs: Examples**

- Usefull abbreviation:
  - \(<\text{any}>\) Φ stands for \(<a_1>\)Φ \(\lor\) \(\cdots\) \(\lor\) \(<a_n>\)Φ
  - \([\text{any}]\) Φ stands for \([a_1)\)Φ \(\land\) \(\cdots\) \(\land\) \([a_n]\)Φ
  - \(<\text{any} - a_1>\) Φ stands for \(<a_2>\)Φ \(\lor\) \(\cdots\) \(\lor\) \(<a_n>\)Φ
  - \([\text{any} - a_1]\) Φ stands for \([a_2]\)Φ \(\land\) \(\cdots\) \(\land\) \([a_n]\)Φ

- Examples:
  - \(<a>\text{true}\) capability of performing action a
  - \([a]\text{false}\) inability of performing action a
  - \(\neg\text{Final} \land <\text{any}>\text{true} \land [\text{any}-a]\text{false}\) necessity/inevitability of performing action a (i.e., action a is the only action possible)
  - \(\neg\text{Final} \land [\text{any}]\text{false}\) deadlock!

**Propositional Dynamic Logic**

- \(Φ := P | \neg Φ | Φ_1 \land Φ_2 | Φ_1 \lor Φ_2 | [r]Φ | <r>Φ\) (atomic propositions) (closed under boolean operators) (modal operators)
- \(r := a | r_1 + r_2 | r_1 ; r_2 | r^* | P^?\) (complex actions as regular expressions)

- Essentially add the capability of expressing partial correctness assertions via formulas of the form
  - \(Φ \rightarrow [r]Φ_2\) under the conditions Φ, all possible executions of r that terminate reach a state of the TS where Φ holds

- Also add the ability of asserting that a property holds in all nodes of the transition system
  - \([a_1 + \cdots + a_n]^*Φ\) in every reachable state of the TS Φ holds

- Useful abbreviations:
  - any stands for \((a_1 + \cdots + a_n)\)
  - u stands for any* Note that + can be expressed also in HM Logic
  - This is the so called master/universal modality
Modal Mu-Calculus

- $\Phi ::= P | \neg \Phi | \Phi_1 \land \Phi_2 | \Phi_1 \lor \Phi_2 | [r]\Phi | <r>\Phi$ (atomic propositions)
- $\mu X.\Phi(X) | v X.\Phi(X)$ (closed under boolean operators)
- $\mu \Phi$ (modal operators)
- It is the most expressive logic of the family of logics of programs.
- It subsumes
  - PDL (modalities involving complex actions are translated into formulas involving fixpoints)
  - LTL (linear time temporal logic)
  - CTS, CTS* (branching time temporal logics)
- Examples:
  - $[\text{any}^*]\Phi$ can be expressed as $v X. \Phi \land [\text{any}]X$
  - $\mu X. \Phi \lor [\text{any}]X$ along all runs eventually $\Phi$
  - $\mu X. \Phi \lor <\text{any}>X$ along some run eventually $\Phi$
  - $v X. [a](\mu Y. <\text{any}>true \land [\text{any}-b]Y) \land X$ every run that that contains $a$ contains later $b$

Model Checking

- Model checking is polynomial in the size of the TS for
  - HennessyMilner Logic
  - PDL
  - Mu-Calculus
- Also model checking is wrt the formula
  - Polynomial for HennessyMilner Logic
  - Polynomial for PDL
  - Polynomial for Mu-Calculus with bounded alternation of fixpoints and $NP \cap coNP$ in general
Model Checking

- Given a TS $T$, one of its states $s$, and a formula $\Phi$ verify whether the formula holds in $s$. Formally:
  \[ T, s \models \Phi \]

- Examples (TS is our vending machine):
  - $S_0 \models \text{Final}$
    - $S_0 \models <10c>\text{true}$ capability of performing action 10c
    - $S_2 \models \text{[big]false}$ inability of performing action big
    - $S_0 \models \text{[10c][big]false}$ after 10c cannot execute big
    - $S_i \models \mu X. \text{Final } \lor [\text{any}] X$ eventually a final state is reached
    - $S_0 \models \lor Z. (\mu X. \text{Final } \lor [\text{any}] X) \land [\text{any}] Z$ or equivalently
      $S_0 \models [\text{any}^*](\mu X. \text{Final } \lor [\text{any}] X)$ from everywhere eventually final

AI Planning as Model Checking

- **Build the TS of the domain:**
  - Consider the set of states formed all possible truth value of the propositions (this works only for propositional setting).
  - Use Pre's and Post of actions for determining the transitions
    Note: the TS is exponential in the size of the description.

- **Write the goal in a logic of program**
  - typically a single least fixpoint formula of Mu-Calculus (compute reachable states intersection states where goal true)

- **Planning:**
  - model check the formula on the TS starting from the given initial state.
  - use the path (paths) used in the above model checking for returning the plan.

- This basic technique works only when we have complete information (or at least total observability on state):
  - Sequential plans if initial state known and actions are deterministic
  - Conditional plans if many possible initial states and/or actions are nondeterministic
**Example**

- **Operators (Services + Mappings)**
  - \(\text{Registered} \land \neg \text{FlightBooked} \rightarrow [S_1: \text{bookFlight}] \text{FlightBooked}\)
  - \(\neg \text{Registered} \rightarrow [S_1: \text{register}] \text{Registered}\)
  - \(\neg \text{HotelBooked} \rightarrow [S_2: \text{bookHotel}] \text{HotelBooked}\)

- **Additional constraints (Community Ontology):**
  - \(\text{TravelSettledUp} \equiv \text{FlightBooked} \land \text{HotelBooked} \land \text{EventBooked}\)

- **Goals (Client Service Requests):**
  - Starting from state \(\text{Registered} \land \neg \text{FlightBooked} \land \neg \text{HotelBooked} \land \neg \text{EventBooked}\)
    \(\text{check} <\text{any*>TravelSettledUp}\)
  - Starting from all states such that \(\neg \text{FlightBooked} \land \neg \text{HotelBooked} \land \neg \text{EventBooked}\)
    \(\text{check} <\text{any*>TravelSettledUp}\)

**Example**

\[
\begin{array}{c}
\text{R} \\
\text{S}_1:f \\
\text{R,F} \\
\text{S}_2:h \\
\text{S}_1:r \\
\text{S}_3:h \\
\text{R,H} \\
\text{S}_1:r \\
\text{H} \\
\text{R,F,T} \\
\end{array}
\]

**Plan:**
\[
\text{S}_1: \text{bookFlight;} \\
\text{S}_2: \text{bookHotel}
\]

Starting from state
\(\text{Registered} \land \neg \text{FlightBooked} \land \neg \text{HotelBooked} \land \neg \text{EventBooked}\)
\(\text{check} <\text{any*>TravelSettledUp}\)
Example

Satisfiability

- Observe that a formula \( \Phi \) may be used to select among all TS TS \( T \) those such that for a given state \( s \) we have that \( T,s \models \Phi \)

- SATISFIABILITY: Given a formula \( \Phi \) verify whether there exists a TS \( T \) and a state \( s \) such that. Formally:

  check whether exists \( T,s \) such that \( T,s \models \Phi \)

- Satisfiability is:
  - PSPACE for HennesyMilner Logic
  - EXPTIME for PDL
  - EXPTIME for Mu-Calculus
References


Composition: the “Roman” Approach
The Roman Approach

Client-formulates the service it requires as a TS using the actions of the community ontology

Available services: described in terms of a TS using actions of the community ontology

The community realizes the client’s target service by “reversing” the mapping and hence using fragments of the computation of the available services.

Community of Services

- A community of Services is
  - a set of services ...
  - ... that share implicitly a common understanding on a common set of actions (common ontology limited to the alphabet of actions)...
  - ... and export their behavior using (finite) TS over this common set of actions

- A client specifies needs as a service behavior, i.e., a (finite) TS using the common set of actions of the community.
(Target & Available) Service TS

- We model services as finite TS $T = (\Sigma, S, s^0, \delta, F)$ with
  - single initial state $(s^0)$
  - deterministic transitions (i.e., $\delta$ is a partial function from $S \times \Sigma$ to $S$)

Note: In this way the client entirely controls/chooses the transition to execute

Example:

$$S_0$$

- $a$: "search by author (and select)"
- $b$: "search by title (and select)"
- $c$: "listen (the selected song)"

Composition: an Example

target service (virtual)

Let's get some intuition of what a composition is through an example

orchestrator

available service 1

available service 2
**Composition: an Example**

**A sample run**

**action request:**

orchestrator response:

---

**Composition: an Example**

**A sample run**

**action request:**

orchestrator response:
**Composition: an Example**

**A sample run**

**action request:** \( a \), \( c \)

**orchestrator response:** \( a,1 \), \( c,1 \)

---

**Composition: an Example**

**A sample run**

**action request:** \( a \), \( c \), \( b \)

**orchestrator response:** \( a,1 \), \( c,1 \), \( b,2 \)
Composition: an Example

A sample run

| action request | a | c | b | c | ...
|---------------|---|---|---|---|---
| orchestrator response | a,1 | c,1 | b,2 | c,2 |

A orchestrator program realizing the target behavior

orchestrator program

**Orchestrator programs**

- **Orchestrator program** is any function \( P(h,a) = i \) that takes a history \( h \) and an action \( a \) to execute and delegates \( a \) to one of the available services \( i \).

- A **history** is the sequence of actions done so far:
  \[ h = a_1 a_2 ... a_k \]

- Observe that to take a decision \( P \) has **full access to the past**, but no access to the future.
  - Note given an history \( h = a_1 a_2 ... a_k \) an the function \( P \) we can reconstruct the state of the target service and of each available service:
    - \( a_1 a_2 ... a_k \) determines the state of the target service
    - \( (a_1, P([], a_1))(a_2, P([a_1], a_2)) ... (a_k, P([a_1 a_2 ... a_{k-1}], a_k)) \) determines the state of each available service.

- **Problem:** synthesize a orchestrator program \( P \) that realizes the target service making use of the available services.

---

**Service Execution Tree**

By "unfolding" a (finite) TS one gets an (infinite) execution tree -- yet another (infinite) TS which bisimilar to the original one.

**Nodes:** history i.e., sequence of actions executed so far.

**Root:** no action yet performed.

**Successor node \( x \cdot a \) of \( x \):** action \( a \) can be executed after the sequence of action \( x \).

**Final nodes:** the service can terminate.
**Alternative (but Equivalent) Definition of Service Composition**

Composition:
- coordinating program ...
- ... that realizes the target service ...
- ... by suitably coordinating available services

⇒ Composition can be seen as:
- a labeling of the execution tree of the **target service** such that ...
- ... each **action** in the execution tree is labeled by the available service that executes it ...
- ... and each possible sequence of actions on the target service execution tree corresponds to possible sequences of actions on the available service execution trees, suitably interleaved

**Example of Composition**

\[ S_0 = \text{orch}( S_1 \parallel S_2 ) \]
**Example of Composition**

\[ S_0 = \text{orch}(S_1 \ || \ S_2) \]

All services start from their starting state

---

**Example of Composition (5)**

\[ S_0 = \text{orch}(S_1 \ || \ S_2) \]

Each action of the target service is executed by at least one of the component services
**Example of composition (6)**

$$S_0 = \text{orch}( S_1 \parallel S_2 )$$

When the target service can be left, then all component services must be in a final state.

**Example of composition (7)**

$$S_0 = \text{orch}( S_1 \parallel S_2 )$$
**Example of composition (8)**

![Diagram of composition](image)

\[ S_0 = \text{orch}( S_1 \parallel S_2 ) \]

Seminari di Ingegneria del Software: integrazione di dati e servizi – aa 2006/07

**Observation**

- This labeled execution tree has a finite representation as a finite TS ...
- ...with transitions labeled by an action and the service performing the action

![Diagram of observation](image)

*Is this always the case when we deal with services expressible as finite TS? See later...*
Questions

Assume services of community and target service are finite TSs

- Can we always check composition existence?
- If a composition exists there exists one which is a finite TS?
- If yes, how can a finite TS composition be computed?

To answer ICSOC’03 exploits PDL SAT

Answers

Reduce service composition synthesis to satisfiability in (deterministic) PDL

- Can we always check composition existence?
  Yes, SAT in PDL is decidable in EXPTIME
- If a composition exists there exists one which is a finite TS?
  Yes, by the small model property of PDL
- How can a finite TS composition be computed?
  From a (small) model of the corresponding PDL formula
**Encoding in PDL**

Basic idea:
- A orchestrator program \( P \) realizes the target service \( T \) iff at each point:
  - \( \forall \) transition labeled \( a \) of the target service \( T \) ...
  - \( \exists \) an available service \( B_i \) (the one chosen by \( P \)) that can make an \( a \)-transition, realizing the \( a \)-transition of \( T \)

- Encoding in PDL:
  - \( \forall \) transition labeled \( a \) ...
  - use **branching**
  - \( \exists \) an available service \( B_i \) that can make an \( a \)-transition ...
  - use underspecified predicates **assigned through SAT**

---

**Structure of the PDL Encoding**

\[
\Phi = \text{Init} \land [u](\Phi_0 \land \bigwedge_{i=1,...,n} \Phi_i \land \Phi_{aux})
\]

- **Initial states of all services**
- **PDL encoding of target service**
- **PDL encoding of \( i \)-th component service**
- **PDL additional domain-independent conditions**

**PDL encoding is polynomial in the size of the service TSs**
**PDL Encoding**

- Target service $S_0 = (\Sigma, S_0, s_0^0, \delta_0, F_0)$ in PDL we define $\Phi_0$ as the conjunction of:
  - $s \rightarrow \neg s'$ for all pairs of distinct states in $S_0$
    - service states are pair-wise disjoint
  - $s \rightarrow <a> T \land [a]s'$ for each $s'=\delta_0(s,a)$
    - target service can do an a-transition going to state $s'$
  - $s \rightarrow [a] \bot$ for each $\delta_0(s,a)$ undef.
    - target service cannot do an a-transition
  - $F_0 = \forall s \in F_0 s$
    - denotes target service final states

- ...

---

**PDL Encoding (cont.d)**

- available services $S_i = (\Sigma, S_i, s_0^i, \delta_i, F_i)$ in PDL we define $\Phi_i$ as the conjunction of:
  - $s \rightarrow \neg s'$ for all pairs of distinct states in $S_i$
    - Service states are pair-wise disjoint
  - $s \rightarrow [a](\text{moved}_i \land s' \lor \neg \text{moved}_i \land s)$ for each $s'=\delta_i(s,a)$
    - if service moved then new state, otherwise old state

  - $s \rightarrow [a](\neg \text{moved}_i \land s)$ for each $\delta_i(s,a)$ undef.
    - if service cannot do a, and a is performed then it did not move
  - $F_i = \forall s \in F_i s$
    - denotes available service final states

- ...

---
PDL Encoding (cont.d)

- Additional assertions $\Phi_{aux}$
  - $<a>T \rightarrow [a] \bigvee_{i=1,..,n} \text{moved}_i$ for each action $a$ at least one of the available services must move at each step
  - $F_0 \rightarrow \bigwedge_{i=1,..,n} F_i$
    when target service is final all comm. services are final
  - $\text{Init} = s_0^0 \land \bigwedge_{i=1,..,n} s_0^i$
    Initially all services are in their initial state

PDL encoding: $\Phi = \text{Init} \land [u](\Phi_0 \land \bigwedge_{i=1,..,n} \Phi_i \land \Phi_{aux})$

Results

Thm[ICSOC’03,IJCIS’05]:
Composition exists iff PDL formula $\Phi$ SAT
From composition labeling of the target service one can build a tree model of the PDL formula and viceversa
Information on the labeling is encoded in predicates moved,

Corollary [ICSOC’03,IJCIS’05]:
Checking composition existence is decidable in EXPTIME

Thm[Muscholl&Walukiewicz FoSSaCS’07]:
Checking composition existence is EXPTIME-hard
**Results on TS Composition**

Thm[ICSOC’03, IJCIS’05]:
If composition exists then finite TS composition exists.

*From a small model of the PDL formula \( \Phi \),
one can build a finite TS machine*

*Information on the output function of the machine is encoded in
predicates moved_1*

\[ \Rightarrow \text{finite TS composition existence of services expressible as}
\text{finite TS is EXPTIME-complete} \]

**Example (1)**

Target service

![Diagram of Target service]

Available services

![Diagram of Available services]

PDL

\[ \cdots 
\]

\[ \cdots 
\]

\[ s_0^0 \land s_1^0 \land s_2^0 
\]

\[ <a> \ T \rightarrow [a] (\text{moved}_1 \lor \text{moved}_2) 
\]

\[ <b> \ T \rightarrow [b] (\text{moved}_1 \lor \text{moved}_2) 
\]

\[ <c> \ T \rightarrow [c] (\text{moved}_1 \lor \text{moved}_2) 
\]

\[ F_0 \rightarrow F_1 \land F_2 
\]
**Example (2)**

Target service

\[ s_0^0 \rightarrow \neg s_0^1 \]
\[ s_0^0 \rightarrow <a> T \land [a] s_0^1 \]
\[ s_0^0 \rightarrow <b> T \land [b] s_0^1 \]
\[ s_0^1 \rightarrow <c> T \land [c] s_0^0 \]
\[ s_0^0 \rightarrow [c] \bot \]
\[ s_0^1 \rightarrow [a] \bot \]
\[ s_0^1 \rightarrow [b] \bot \]
\[ F_0 = s_0^0 \]

\[ \ldots \]
\[ \ldots \]
\[ \ldots \]

---

**Example (3)**

Available services

\[ s_1^0 \rightarrow \neg s_1^1 \]
\[ s_1^0 \rightarrow [a] (moved_1 \land s_1^1 \lor \neg moved_1 \land s_1^0) \]
\[ s_1^0 \rightarrow [c] \neg moved_1 \land s_1^0 \]
\[ s_1^0 \rightarrow [b] \neg moved_1 \land s_1^0 \]
\[ s_1^1 \rightarrow [a] \neg moved_1 \land s_1^1 \]
\[ s_1^1 \rightarrow [b] \neg moved_1 \land s_1^1 \]
\[ s_1^1 \rightarrow [c] (moved_1 \land s_1^0 \lor \neg moved_1 \land s_1^0) \]
\[ F_1 = s_1^0 \]

\[ s_2^0 \rightarrow \neg s_2^1 \]
\[ s_2^0 \rightarrow [b] (moved_2 \land s_2^1 \lor \neg moved_2 \land s_2^0) \]
\[ s_2^0 \rightarrow [c] \neg moved_2 \land s_2^0 \]
\[ s_2^0 \rightarrow [a] \neg moved_2 \land s_2^0 \]
\[ s_2^1 \rightarrow [b] \neg moved_2 \land s_2^1 \]
\[ s_2^1 \rightarrow [a] \neg moved_2 \land s_2^1 \]
\[ s_2^1 \rightarrow [c] (moved_2 \land s_2^0 \lor \neg moved_2 \land s_2^0) \]
\[ F_2 = s_2^0 \]

\[ \ldots \]
**Example (4)**

Check: run SAT on PDL formula $\Phi$

---

**Example**

Check: run SAT on PDL formula $\Phi$

Yes $\Rightarrow$ (small) model
Example

Check: run SAT on PDL formula $\Phi$
Yes $\Rightarrow$ (small) model

$\Rightarrow$ extract finite TS

Example

Check: run SAT on PDL formula $\Phi$
Yes $\Rightarrow$ (small) model

$\Rightarrow$ extract finite TS
$\Rightarrow$ minimize finite TS  
(similar to Mealy machine minimization)
**Results on Synthesizing Composition**

- Using PDL reasoning algorithms based on model construction (cf. tableaux), build a (small) model
  
  *Exponential in the size of the PDL encoding/services finite TS*

  *Note: SitCalc, etc. can compactly represent finite TS, PDL encoding can preserve compactness of representation*

- From this model extract a corresponding finite TS
  
  *Polynomial in the size of the model*

- Minimize such a finite TS using standard techniques (opt.)
  
  *Polynomial in the size of the TS*

  *Note: finite TS extracted from the model is not minimal because encodes output in properties of individuals/states*

**Tools for Synthesizing Composition**

- In fact we use only a fragment of PDL in particular we use fixpoint (transitive closure) only to get the universal modality ...

- ... thanks to a tight correspondence between PDLs and Description Logics (DLs), we can use current highly optimized DL reasoning systems to do synthesis ...

- ... when the ability or returning models will be added ...

  *Pellet already has this ability, and we are exploring its use*

- ... meanwhile we have developed a prototype tool on this idea (see last Massimo’s lecture)
**Composition via Simulation**

**Bisimulation**

- A binary relation $R$ is a **bisimulation** iff:

  $$(s,t) \in R \text{ implies that}$$

  - $s$ is **final** iff $t$ is **final**
  - for all actions $a$
    - if $s \xrightarrow{a} s'$ then $\exists t' . t \xrightarrow{a} t'$ and $(s',t') \in R$
    - if $t \xrightarrow{a} t'$ then $\exists s' . s \xrightarrow{a} s'$ and $(s',t') \in R$

- A state $s_0$ of transition system $S$ is **bisimilar**, or simply **equivalent**, to a state $t_0$ of transition system $T$ iff there exists a **bisimulation** between the initial states $s_0$ and $t_0$.

- Notably
  - **bisimilarity** is a bisimulation
  - **bisimilarity** is the **largest** bisimulation

  *Note it is a co-inductive definition!*
Computing Bisimilarity on Finite Transition Systems

Algorithm ComputingBisimulation
Input: transition system $\mathcal{S}_S = < A, S, S_0, \delta_S, F_S>$ and transition system $\mathcal{S}_T = < A, T, T_0, \delta_T, F_T>$
Output: the bisimilarity relation (the largest bisimulation)

Body
$R = \emptyset$
$R' = S \times T - \{(s,t) | \neg(s \in F_S \equiv t \in F_T)\}$
while ($R \neq R'$) {
    $R := R'$
    $R' := R' - \{(s,t) | \exists s',a. s \rightarrow_a s' \land \exists t'. t \rightarrow_a t' \land (s',t') \in R'\}$
    $\{(s,t) | \exists t',a. t \rightarrow_a t' \land \exists s'. s \rightarrow_a s' \land (s',t') \in R'\}$
}
return $R'$

Ydob

Simulation

- A binary relation $R$ is a simulation iff:

  $(s,t) \in R$ implies that
  - $s$ is final implies that $t$ is final
  - for all actions $a$
    - if $s \rightarrow_a s'$ then $\exists t'. t \rightarrow_a t'$ and $(s',t') \in R$

- A state $s_0$ of transition system $S$ is simulated by a state $t_0$ of transition system $T$ iff there exists a simulation between the initial states $s_0$ and $t_0$.

- Notably
  - simulated-by is a simulation
  - simulated-by is the largest simulation

  Note it is a co-inductive definition!

- NB: A simulation is just one of the two directions of a bisimulation
Computing Simulation on Finite Transition Systems

**Algorithm** ComputingSimulation

**Input:** transition system $TS_S = < A, S, S^0, \delta_S, F_S >$ and transition system $TS_T = < A, T, T^0, \delta_T, F_T >$

**Output:** the simulated-by relation (the largest simulation)

**Body**

1. $R = \emptyset$
2. $R' = S \times T - \{(s, t) | s \in F_S \land \neg t \in F_T \}$
3. while ($R \neq R'$) {
   1. $R' := R'$
   2. $R' := R' - \{(s, t) | \exists s', a \cdot s \rightarrow_a s' \land \neg \exists t'. t \rightarrow_a t' \land (s', t') \in R' \}$
4. return $R'$

$Ydob$

Potential Behavior of the Whole Community

- Let $TS_1, \cdots, TS_n$ be the TSs of the component services.

- The **Community TS** is defined as the asynchronous product of $TS_1, \cdots, TS_n$, namely:
  
  $TS_c = < A, S_c, S^0_c, \delta_c, F_c >$ where:
  
  - $A$ is the set of actions
  - $S_c = S_1 \times \cdots \times S_n$
  - $S^0_c = \{(s^0_1, \cdots, s^0_m)\}$
  - $F \subseteq F_1 \times \cdots \times F_n$
  - $\delta_c \subseteq S_c \times A \times S_c$ is defined as follows:
    
    $(s_1 \times \cdots \times s_n) \rightarrow_a (s'_1 \times \cdots \times s'_n)$ iff
    
    1. $\exists i. s_i \rightarrow_a s'_i \in \delta_i$
    2. $\forall j \neq i. s'_j = s_j$
**Example of Composition**

- Available Services

![Diagram of Available Services](null)

- Target Service

![Diagram of Target Service](null)

**Example of Composition**

Community TS

![Diagram of Community TS](null)

Target Service

![Diagram of Target Service](null)

**Composition exists!**
Composition via Simulation

- **Thm[Subm07]**
  A composition realizing a target service TS exists if there exists a simulation relation between the initial state $s_i^0$ of TS and the initial state $(s_1^0, ..., s_n^0)$ of the community TS.

- Notice if we take the union of all simulation relations, then we get the largest simulation relation $S$, still satisfying the above condition.

- **Corollary[Subm07]**
  A composition realizing a target service TS exists iff $(s_i^0, (s_1^0, ..., s_n^0)) \in S$.

- **Thm[Subm07]**
  Computing the largest simulation $S$ is polynomial in the size of the target service TS and the size of the community TS...

  ... hence it is **EXPTIME** in the size of the available services.

Composition via Simulation

- Given the largest simulation $S$ form TS to TS (which include the initial states), we can build the **orchestrator generator**.

- This is an orchestrator program that can change its behavior reacting to the information acquired at run-time.

- **Def:** $OG = < A, [1,...,n], S, s_i^0, \omega, \delta, F_r >$ with
  - $A$: the **actions** shared by the community
  - $[1, ..., n]$: the **identifiers** of the available services in the community
  - $S = S_i \times S_1 \times ... \times S_n$: the **states** of the orchestrator program
  - $s_i^0 = (s_i^0, s_1^0, ..., s_n^0)$: the **initial state** of the orchestrator program
  - $F_r \subseteq \{ (s_i, s_1, ..., s_n) \mid s_i \in F_i \}$: the **final states** of the orchestrator program
  - $\omega_i : S_i \times A_i \rightarrow [1,...,n]$ : the **service selection function**, defined as follows:
    - If $s_i \rightarrow s_i'$ then
      
      **chose** $k$ s.t. $s_k' \rightarrow s_k \rightarrow (s_i', s_1, ..., s_n)$ \( \in S \)
  
  - $\delta : S_i \times A \times [1,...,n] \rightarrow S_i$ : the **state transition function**, defined as follows:
    - Let $\omega(s_i, s_1, ..., s_n, a) = k$ then
      
      $(s_i, s_1, ..., s_n) \rightarrow_{a,k} (s_i', s_1, ..., s_n)$ where $s_k \rightarrow s_k'$
**Composition via Simulation**

- For generating OG we need only to compute $S$ and then apply the template above.

- For running an orchestrator from the OG we need to store and access $S$ (*polynomial time, exponential space*) ...

- ... and compute $\omega_r$ and $\delta_r$ at each step (*polynomial time and space*).

**Extension to the Roman Model**
Extensions

- **Nondeterministic (angelic) target** specification
  - Loose specification in client request
  - Angelic (don’t care) vs devilish (don’t know) nondeterminism
  - See [ICDCS’04]

- **Nondeterministic (devilish) available services**
  - Incomplete specification in available services
  - Devilish (don’t know) vs angelic (don’t care) nondeterminism
  - See below & [IJCAI’07]

- **Distributing** the orchestration
  - Often a centralized orchestration is unrealistic: eg. services deployed on mobile devices
    - too tight coordination
    - too much communication
    - orchestrator cannot be embodied anywhere
  - Drop centralized orchestrator in favor of **independent controllers** on single available services (exchanging messages)
  - Under suitable conditions: a distributed orchestrator exists iff a centralized one does
  - Still decidable (EXPTIME-complete)
  - See [AAAI’07]

- **Dealing with data**
  - This is the single most difficult issue to tackle
    - First results: actions as DB updates, see [VLDB’05]
    - Literature on Abstraction in Verification
  - From finite to **infinite transition** systems!

- **Security and trust** aware composition [SWS’06]

- **Automatic Workflows** Composition of Mobile Services [ICWS’07]

---

**Nondeterministic Available Services**
Nondeterminism in Available Services

- Nondeterministic available services
  - Incomplete information on the actual behavior
  - Mismatch between behavior description (which is in terms of the environment actions) and actual behavior of the agents/devices

- Deterministic target service
  - It's a spec of a desired service: (devilish) nondeterminism is banned

In general, devilish nondeterminism difficult to cope with eg. nondeterminism moves AI Planning from PSPACE (classical planning) to EXPTIME (contingent planning with full observability [Rintanen04])

Example: Nondeterministic Available Services
**Example: Nondeterministic Available Services**

Seminari di Ingegneria del Software: integrazione di dati e servizi –aa 2006/07

Giuseppe De Giacomo

95

---

**Available Services**

orchestrator
Example: Nondeterministic Available Services

target service

observe the actual state!

service 1

service 2

Example: Nondeterministic Available Services

target service

observe the actual state!

service 1

service 2
Example: Nondeterministic Available Services

target service

service 1

observe the actual state!

service 2

orchestrator

An Orchestrator Program Realizing the Target Service

target service

service 1

orchestrator program

orchestrator

service 2
**Orchestrator Programs**

- **Orchestrator program** is any function \( P(h,a) = i \) that takes a **history** \( h \) and an **action** \( a \) to execute and delegates \( a \) to one of the available services \( i \).

- A **history** is a sequence of the form:
  \[
  (s_1^0,s_2^0,\ldots,s_n^0,e^0) \ a_1 (s_1^1,s_2^1,\ldots,s_n^1,e^1) \ldots \ a_k (s_1^k,s_2^k,\ldots,s_n^k,e^k)
  \]

- Observe that to take a decision \( P \) has **full access to the past**, but no access to the future.

- **Problem**: synthesize a **orchestrator program** \( P \) that realizes the target service making use of the available services.

---

**Technique: Reduction to PDL**

Basic idea:

- A orchestrator program \( P \) realizes the target service \( T \) iff at each point:
  - \( \forall \) a transition labeled \( a \) of the target service \( T \) ...
  - ... \( \exists \) an available service \( B_i \) (the one chosen by \( P \)) which can make an \( a \)-transition ...
  - ... and \( \forall a \)-transition of \( B_i \) realize the \( a \)-transition of \( T \)

- **Encoding in PDL**:
  - \( \forall \) transition labeled \( a \) ...
    use branching
  - \( \exists \) an available service \( B_i \) ...
    use unspecified predicates assigned through SAT
  - \( \forall a \)-transition of \( B_i \) ...
    use branching again
Technical Results: Theoretical

**Thm [IJCAI’07]** Checking the existence of orchestrator program realizing the target service is **EXPTIME-complete**.

*EXPTIME-hardness due to Muscholl&Walukiewicz07 for deterministic services*

**Thm [IJCAI’07]** If a orchestrator program exists there exists one that is **finite state**.

*Exploits the finite model property of PDL*

*Note: same results as for deterministic services!*

Technical Results: Practical

Reduction to PDL provides also a practical sound and complete technique to compute the orchestrator program also in this case

- Use state-of-the-art tableaux systems for OWL-DL for checking SAT of PDL formula $\Phi$ coding the composition existence
- **If SAT, the tableau returns a finite model of $\Phi$**
- exponential in the size of the behaviors
- Project away irrelevant predicates from such model, and possibly minimize
- **The resulting structure is a finite orchestrator program that realizes the target behavior**
- polynomial in the size of the model

*eg, PELLET @ Univ. Maryland*
Nondeterministic Available Services: Composition à la Simulation

Composition à la Simulation

- We consider binary relations $R$ satisfying the following co-inductive condition:

  $$(s,(q_1, \ldots, q_n)) \in R \text{ implies that }$$
  $$- \text{ if } s \text{ is final then } q_i \text{ with } i=1, \ldots, n \text{ is final}$$
  $$- \text{ for all actions } a$$
  $$\quad \text{ if } s \rightarrow s' \text{ then } \exists k \in 1..n.$$  
  $$\quad \quad \exists q_k' \cdot q_k \rightarrow a q_k' \supset (s',(q_1',\ldots,q_n')) \in R$$

  Note similar in the spirit to simulation relation!
  But more involved, since it deals with

  - the existential choice (as the simulation) of the service, and
  - the universal condition on the nondeterministic branches!

- A composition realizing a target service $TS$ exists if there exists a relation $R$ satisfying the above condition between the initial state $s_0$ of $TS$ and the initial state $(s_1^0, \ldots, s_n^0)$ of the community big $TS_c$.

- Notice if we take the union of all such relation $R$ then we get the largest relation $RR$ satisfying the above condition.

- A composition realizing a target service $T$ exists iff $(s_0^0, (s_1^0, \ldots, s_n^0)) \in RR$. 

Seminari di Ingegneria del Software: integrazione di dati e servizi – aa 2006/07
Giuseppe De Giacomo 106
Composition à la Simulation

- Given \( RR \) form \( TS_i \) to \( TS_j \)(which include the initial states), we can build the orchestrator generator.
- This is an orchestrator program that can change its behavior reacting to the information acquired at run-time.
- Def: \( OG = <A, [1, ..., n], S, s_0, \omega, \delta, F_r> \) with
  - \( A \) : the actions shared by the community
  - \([1, ..., n]\) : the identifiers of the available services in the community
  - \( S = S_i \times S_i \times \cdots \times S_m \) : the states of the orchestrator program
  - \( s_i \) : \( (s_i^0, s_i^1, ..., s_i^m) \) : the initial state of the orchestrator program
  - \( F_r \subset \{ (s_i, s_1, ..., s_n) \mid s_i \in F_i \} \) : the final states of the orchestrator program
- \( \omega : S_i \times A \rightarrow [1, ..., n] \) : the service selection function, defined as follows:
  - If \( s_i \rightarrow s'_i \) then
    - \( \omega \) s.t. \( s_k \rightarrow s_k', s_k \rightarrow s_k'' \wedge s_k \rightarrow s_k' \Rightarrow (s_k', s_k ..., s_k') \in RR \)
- \( \delta \subseteq S_i \times A \times [1, ..., n] \times S_i \) : the state transition relation, defined as follows:
  - Let \( \omega(s_i, s_i, ..., s_i, s_i, a) = k \) then
    \( (s_i, s_i ..., s_i, s_i) \rightarrow_{\delta} (s_i', s_i' ..., s_i') \) for each \( s_i \rightarrow_{\delta} s_i' \)

Composition à la Simulation

- Computing \( RR \) is polynomial in the size of the target service \( TS \) and the size of the community \( TS \)
- ... composition can be done in \( \text{EXPTIME} \) in the size of the available services

- For generating \( OG \) we need only to compute \( RR \) and then apply the template above
- For running the \( OG \) we need to store and access \( RR \) (polynomial time, exponential space) ...
- ... and compute \( \omega \) \( r \) and \( \delta \) \( r \) at each step (polynomial time and space)
**Example of Composition**

Available Services

```
TS_1
  a
  b

TS_2
  b
```

Target Service

```
TS_i
  a
  b
```

**Example of Composition**

Community TS

```
TS_c
  a
  b

TS_i
  a
  b
```

Target Service

```
TS_i
  a
  b
```

*Composition exists!*
References


[ICSOC'04] Daniela Berardi, Giuseppe De Giacomo, Maurizio Lenzerini, Massimo Meccella, Diego Calvanese: Synthesis of underspecified composite e-services based on automated reasoning. ICSOC 2004: 105-114


[ICSOC'05] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Massimo Meccella: Composition of Services with Nondeterministic Observable Behavior. ICSOC 2005: 520-526


