Algebraic simulations

Narciso Martí-Oliet
(joint work with José Meseguer and Miguel Palomino)

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
`narciso@sip.ucm.es`

IFIP WG 1.3
Braga, Portugal
March 23, 2007

Introduction

- The Maude system includes a *model checker* to prove temporal properties of systems.
- In many cases it is necessary to *abstract* a system in order to obtain another system with a small enough number of states.
- In other cases we have to provide more *concrete* details in the specification of a system, for example when *refining* or *implementing* a specification.
- In general we need to have concepts and methods justifying that a system *simulates* another.

Goals

- *Generalize* the notion of simulation between computational systems as much as possible.
- Provide general *representability results* of simulations in *rewriting logic*, an executable framework with good properties for representing concurrent systems.
- Simulations are essential for *compositional reasoning*.
- Simulations *reflect* interesting classes of temporal logic properties.
- The more general the notion, the wider its applicability.
- Representability in rewriting logic is motivated by *ease of specification*, because it is a flexible framework, and *executability*.

Representing computational systems

- The behavior of state-based systems is represented by means of *transition systems* $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$, where
  - A is a set of states, and
  - $\rightarrow_{\mathcal{A}} \subseteq A \times A$ is a binary relation called the transition relation.
- To reason about system properties it is necessary to say which atomic propositions hold in a state.
  A *Kripke structure* is a triple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$, where
  - $(A, \rightarrow_{\mathcal{A}})$ is a transition system such that $\rightarrow_{\mathcal{A}}$ is a total relation, and
  - $L_{\mathcal{A}} : A \rightarrow \mathcal{P}(AP)$ is a labelling function associating each state with the set of atomic properties in AP that it satisfies.

Relating Systems

A simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ should:

- *Reflect* interesting properties: if something is true in $\mathcal{B}$ it must also hold in $\mathcal{A}$.
  1. It provides a way of showing that implementation $\mathcal{A}$ satisfies specification $\mathcal{B}$.
  2. It allows to study properties of specification $\mathcal{A}$ in a simpler system, or abstraction, $\mathcal{B}$.
- Be *compositional*.

Basic simulations

- Given two transition systems $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$, a *simulation of transition systems* $H : \mathcal{A} \longrightarrow \mathcal{B}$ is a binary relation $H \subseteq A \times B$ such that if $a \rightarrow_{\mathcal{A}} a'$ and aHb then there exists $b' \in B$ with $b \rightarrow_{\mathcal{B}} b'$ and $a'Hb'$.

$$\begin{array}{ccc} a & \longrightarrow_{\mathcal{A}} & a' \\ H & & H \\ b & \longrightarrow_{\mathcal{B}} & b' \end{array}$$

- Given two Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ over the same set AP of atomic propositions, an *AP-simulation* $H : \mathcal{A} \longrightarrow \mathcal{B}$ of $\mathcal{A}$ by $\mathcal{B}$ is a simulation $H : (A, \rightarrow_{\mathcal{A}}) \longrightarrow (B, \rightarrow_{\mathcal{B}})$ of transition systems such that if aHb then $L_{\mathcal{B}}(b) = L_{\mathcal{A}}(a)$.

Reflection of properties

- An AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ *reflects* the satisfaction of a formula $\varphi \in \mathrm{CTL}^*(AP)$ when either
  - φ is a state formula and then $\mathcal{B}, b \models \varphi$ and aHb imply $\mathcal{A}, a \models \varphi$; or
  - φ is a path formula and then $\mathcal{B}, \rho \models \varphi$ and $\pi H \rho$ imply $\mathcal{A}, \pi \models \varphi$.

Theorem (reflection theorem)

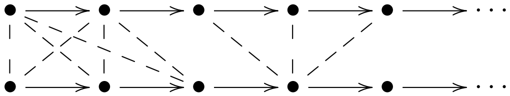
AP-simulations reflect the satisfaction of all formulas in the logic $\mathrm{ACTL}^*(AP)$.

Generalizing simulations

- By slightly restricting the logic, the definition can be generalized.
- If *negations* are not allowed it is enough to require:

$$aHb \text{ implies } L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$$

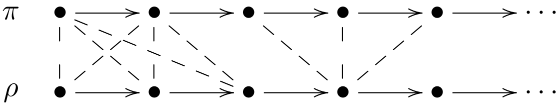
  (Notice that negations can always be pushed to the atoms and be replaced by additional propositions.)
- The requirement that transitions can be mimicked is too strong when we try to relate systems of different granularity. If the *Next* operator is forbidden, transitions need be mimicked only up to *stuttering*.



Stuttering simulations

- Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ be transition systems and $H \subseteq A \times B$ a relation. Given a path π in $\mathcal{A}$ and a path ρ in $\mathcal{B}$, we say that ρ *H-matches* π if there are strictly increasing functions $\alpha, \beta : \mathbb{N} \longrightarrow \mathbb{N}$ with $\alpha(0) = \beta(0) = 0$ such that, for all $i, j, k \in \mathbb{N}$, if $\alpha(i) \leq j < \alpha(i+1)$ and $\beta(i) \leq k < \beta(i+1)$, then $\pi(j) H \rho(k)$.

- **Example**: the beginning of two matching paths, with broken lines meaning related elements, and where $\alpha(0) = \beta(0) = 0$, $\alpha(1) = 2$, $\beta(1) = 3$, $\alpha(2) = 5$, $\beta(2) = 4$, etc.



Stuttering simulations

- Given two transition systems $\mathcal{A}$ and $\mathcal{B}$, a *stuttering simulation of transition systems* $H : \mathcal{A} \longrightarrow \mathcal{B}$ is a binary relation $H \subseteq A \times B$ such that if aHb then for each path π in $\mathcal{A}$ beginning in a there exists a path ρ in $\mathcal{B}$ beginning in b which H-matches π.
- Given two Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ over AP, a *stuttering* AP*-simulation* $H : \mathcal{A} \longrightarrow \mathcal{B}$ is a stuttering simulation of transition systems $H : (A, \rightarrow_{\mathcal{A}}) \longrightarrow (B, \rightarrow_{\mathcal{B}})$ such that if aHb then $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.

Well-founded simulations

- Let $\mathcal{A} = (A, \to_{\mathcal{A}})$ and $\mathcal{B} = (B, \to_{\mathcal{B}})$ be transition systems. A relation $H \subseteq A \times B$ is a *well-founded simulation of transition systems* from $\mathcal{A}$ to $\mathcal{B}$ if there exist functions $\mu : A \times B \longrightarrow W$ and $\mu' : A \times A \times B \longrightarrow \mathbb{N}$, with $(W, <)$ a well-founded order, such that if aHb and $a \to_{\mathcal{A}} a'$, then either
  - there exists b' such that $b \to_{\mathcal{B}} b'$ and $a'Hb'$, or
  - $a'Hb$ and $\mu(a', b) < \mu(a, b)$, or
  - there exists b' such that $b \to_{\mathcal{B}} b'$, aHb', and $\mu'(a, a', b') < \mu'(a, a', b)$.
- Notice that when H is a function only the first two conditions are applicable, and in such case the function μ' can be dispensed with.

Well-founded simulations

- Given two Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ over AP, a relation $H \subseteq A \times B$ is a *well-founded AP-simulation* if H is a well-founded simulation of transition systems and in addition aHb implies $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.

Theorem (Manolios)

Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ be two Kripke structures over AP and $H \subseteq A \times B$. Then, H is a well-founded AP-simulation if and only if it is a stuttering AP-simulation.

Reflection of properties

- A stuttering AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ *reflects* the satisfaction of a formula $\varphi \in \mathrm{CTL}^*(AP)$ when either
  - φ is a state formula and then $\mathcal{B}, b \models \varphi$ and aHb imply $\mathcal{A}, a \models \varphi$; or
  - φ is a path formula and then $\mathcal{B}, \rho \models \varphi$ and ρ H-matches π imply $\mathcal{A}, \pi \models \varphi$.

Theorem (reflection theorem)

Stuttering AP-simulations reflect the satisfaction of all formulas in the logic $\mathrm{ACTL}^* \setminus \{\neg, \mathbf{X}\}(AP)$ *(i.e., formulas not containing negation or next operators).*

Shifting our ground

- A third generalization consists in relating systems over different sets of atomic propositions: *shifting our ground*.
- Given a Kripke structure $\mathcal{A}$ over a set AP and another $\mathcal{B}$ over a set AP', a simulation $(\alpha, H) : (AP, \mathcal{A}) \longrightarrow (AP', \mathcal{B})$ consists of:
  - a function $\alpha : AP \longrightarrow \mathrm{State}\backslash\neg(AP')$, and
  - an AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}|_\alpha$

  where $\mathcal{B}|_\alpha = (B, \rightarrow_\mathcal{B}, L_{\mathcal{B}|_\alpha})$, the "restriction" of $\mathcal{B}$ to AP', is such that $L_{\mathcal{B}|_\alpha} = \{p \in AP \mid \mathcal{B}, b \models \alpha(p)\}$.
- After considering all these generalizations, appropriate sets of properties are still reflected.

A categorical hierarchy

- All these different notions of simulation give rise to increasingly more general categories:
  - For transition systems: **STSys**.
  - For the basic simulations: $\mathbf{KSim}_{AP}$.
  - For stuttering simulations: $\mathbf{KSSim}_{AP}$.
  - For the most general ones: **KSSim**. This category can alternatively be obtained with the Grothendieck construction through all the different $\mathbf{KSSim}_{AP}$.
- There are corresponding subcategories for simulation maps, bisimulations, etc.

Ingredients of rewriting logic

- Types and subtypes.
- Typed operators providing syntax: *signature* Σ.
- Syntax allows the construction of both static data and states: term algebra T_Σ.
- *Equations* E define functions over static data as well as properties of states.
- *Rewrite rules* R define transitions between states.
- Deduction in the logic corresponds to computation with those transitions.
- The *Maude* language is an implementation of rewriting logic, allowing the execution of specifications satisfying some admissibility requirements.

Kripke structure defined by a rewrite system $\mathcal{R} = (\Sigma, E, R)$

- States are the terms $T_{\Sigma/E,k}$ in the equational theory (Σ, E) with a distinguished type k.
- Transitions are defined from the rules in R: a transition consists in applying a rewrite rule to a unique subterm of the source state.
- The transition system asociated to $\mathcal{R}$ and k is denoted by $\mathcal{T}(\mathcal{R})_k$.
- We add state predicates Π defined by means of equations D in an equational theory $(\Sigma', E \cup D)$ conservatively extending (Σ, E).
- The corresponding Kripke structure is denoted by $\mathcal{K}(\mathcal{R}, k)_\Pi$.

Simulations in rewriting logic

We consider four increasingly more general ways of defining simulations in rewriting logic:

- *Equational abstractions*: just add new equations, say E', to the specification of the system of interest (Σ, E, R) to get a quotient $(\Sigma, E \cup E', R)$.
- Instead of theory inclusions $(\Sigma, E) \subseteq (\Sigma', E')$, use arbitrary *theory interpretations* $H : (\Sigma, E) \longrightarrow (\Sigma', E')$.
- Simulation *maps* as *equationally defined* functions in an extension of the disjoint union of the rewrite theories that specify the systems.
- Simulations given by *rewrite relations*, in the same extension.

Simulations in rewriting logic: Previous papers

- J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In F. Baader, editor, *Automated Deduction - CADE-19*, LNCS 2741, pages 2–16. Springer, 2003.
- J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. Extended version. Submitted. `http://maude.sip.ucm.es/~miguelpt`, 2007.
- N. Martí-Oliet, J. Meseguer, and M. Palomino. Theoroidal maps as algebraic simulations. In J. L. Fiadeiro et al., editors, *Recent Trends in Algebraic Development Techniques, WADT 2004*, LNCS 3423, pages 126–143. Springer, 2005.
- M. Palomino, J. Meseguer, and N. Martí-Oliet. A categorical approach to Kripke structures and simulations. In J. L. Fiadeiro et al., editors, *Algebra and Coalgebra in Computer Science, CALCO 2005*, LNCS 3629, pages 313–330. Springer, 2005.

Simulation maps as equationally defined functions

We define a category **SRWTh**:

- Objects are pairs $(\mathcal{R}, k)$, with $\mathcal{R}$ a rewrite theory and k a distinguished type in $\mathcal{R}$.
- A morphism $(\mathcal{R}_1, k_1) \longrightarrow (\mathcal{R}_2, k_2)$ in **SRWTh**, called an *algebraic stuttering map of transition systems*, is a stuttering map $h : \mathcal{T}(\mathcal{R}_1)_{k_1} \longrightarrow \mathcal{T}(\mathcal{R}_2)_{k_2}$ such that there exists a theory extension (Ω, G) containing the equational parts of $\mathcal{R}_1$ and $\mathcal{R}_2$ in which h can be equationally defined through an operator $h : k_1' \longrightarrow k_2'$ (where the primes indicate the corresponding names for the disjoint copies of the kinds).

Simulation maps as equationally defined functions

We define a functor $\mathcal{T} : \mathbf{SRWTh} \longrightarrow \mathbf{STSys}$ as follows:

- for objects $(\mathcal{R}, k)$,

$$\mathcal{T}(\mathcal{R}, k) = \mathcal{T}(\mathcal{R})_k$$

- for morphisms $h : (\mathcal{R}_1, k_1) \longrightarrow (\mathcal{R}_2, k_2)$,

$$\mathcal{T}(h) = h$$

Theorem

The functor $\mathcal{T} : \mathbf{SRWTh} \longrightarrow \mathbf{STSys}$ is surjective on objects, full, and faithful, with the obvious restriction for non-stuttering maps.

Simulation maps as equationally defined functions

Objects in $\mathbf{SRWTh}_{\models}$ are given by triples $(\mathcal{R}, (\Sigma', E \cup D), J)$ where:

1. $\mathcal{R} = (\Sigma, E, R)$ is a rewrite theory specifying the transition system.
2. $(\Sigma, E) \subseteq (\Sigma', E \cup D)$ is a protecting theory extension, containing and protecting also the theory *BOOL* of Booleans, that defines the atomic propositions satisfied by the states. We define $\Pi \subseteq \Sigma'$ as the subsignature of operators of coarity *Prop*.
3. $J : BOOL_{\models} \longrightarrow (\Sigma', E \cup D)$ is a membership equational theory morphism that selects the distinguished type of states $J(State)$, and such that: (i) it is the identity when restricted to *BOOL*, (ii) $J(Prop) = Prop$, and (iii) $J(\_ \models \_ : State\ Prop \rightarrow Bool) = \_ \models \_ : J(State)\ Prop \rightarrow Bool$.

Simulation maps as equationally defined functions

A morphism $(\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ in $\mathbf{SRWTh}_{\models}$, called an *algebraic stuttering map*, is a pair (α, h) such that:

1. $(\alpha, h) : \mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1} \longrightarrow \mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$ is a stuttering map of Kripke structures.
2. There exists a theory extension (Ω, G) containing and protecting disjoint copies of $(\Sigma'_1, E_1 \cup D_1)$ and $(\Sigma'_2, E_2 \cup D_2)$ in which α and h can be equationally defined through operators $\alpha : Prop_1 \longrightarrow StateForm_2$ and $h : J_1(State)_1 \longrightarrow J_2(State)_2$ in Ω; the subscripts $1, 2$ indicate the corresponding names for the disjoint copies of the kinds, and $StateForm_2$ is a new kind for representing state formulas over $Prop_2$.

Simulation maps as equationally defined functions

The construction that associates a Kripke structure to a rewrite theory is a functor. We define $\mathcal{K} : \mathbf{SRWTh}_{\models} \longrightarrow \mathbf{KSMap}$ as:

- for objects $(\mathcal{R}, (\Sigma', E \cup D), J)$,

$$\mathcal{K}(\mathcal{R}, (\Sigma', E \cup D), J) = \mathcal{K}(\mathcal{R}, J(State))_{\Pi}$$

- for morphisms $(\alpha, h) : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$,

$$\mathcal{K}(\alpha, h) = (\alpha, h)$$

Theorem (representability)

The functor $\mathcal{K} : \mathbf{SRWTh}_{\models} \longrightarrow \mathbf{KSMap}$ *is surjective on objects, full, and faithful, with the obvious restrictions for non-stuttering maps.*

Example: A simple functional language

- We consider a simple functional language *Fpl*.
- Syntactic categories:

$$\begin{array}{rcl} op & \in & Op \\ x & \in & Var \\ e & \in & Exp \\ n & \in & Num \end{array} \qquad \begin{array}{rcl} bop & \in & BOp \\ bx & \in & BVar \\ be & \in & BExp \end{array}$$

- Grammar (signature in the algebraic representation):

$$\begin{array}{rcl} op & ::= & + \mid - \mid * \\ bop & ::= & \mathsf{And} \mid \mathsf{Or} \\ e & ::= & n \mid x \mid e' \; op \; e'' \mid \mathsf{If}\ be\ \mathsf{Then}\ e'\ \mathsf{Else}\ e'' \mid \mathsf{let}\ x = e'\ \mathsf{in}\ e'' \\ be & ::= & bx \mid \mathsf{T} \mid \mathsf{F} \mid be' \; bop \; be'' \mid \mathsf{Not}\ be' \mid \mathsf{Equal}(e, e') \end{array}$$

An operational semantics for *Fpl*

- A *state* in the operational semantics is a pair $\langle \rho, e \rangle$, where ρ is an environment assigning values to variables and e is an *Fpl* expression.
- A final state is a pair $\langle \rho, v \rangle$, where v is a value, i.e., either a number n or a boolean constant T or F.
- The operational semantics defines a step in the evaluation of an expression
$$\langle \rho, e \rangle \rightarrow_A \langle \rho', e' \rangle$$
- These steps are repeated until the final value of a given expression is obtained.

An operational semantics for *Fpl*
(some rules)

Var $$\frac{}{\langle \rho, x\rangle \rightarrow_A \langle \rho, \rho(x)\rangle}$$

Op $$\frac{}{\langle \rho, v\ op\ v'\rangle \rightarrow_A \langle \rho, Ap(op, v, v')\rangle}$$

$$\frac{\langle \rho, e\rangle \rightarrow_A \langle \rho', e''\rangle}{\langle \rho, e\ op\ e'\rangle \rightarrow_A \langle \rho', e''\ op\ e'\rangle} \qquad \frac{\langle \rho, e'\rangle \rightarrow_A \langle \rho', e''\rangle}{\langle \rho, e\ op\ e'\rangle \rightarrow_A \langle \rho', e\ op\ e''\rangle}$$

If $$\frac{\langle \rho, be\rangle \rightarrow_B \langle \rho', be'\rangle}{\langle \rho, \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e'\rangle \rightarrow_A \langle \rho', \mathsf{If}\ be'\ \mathsf{Then}\ e\ \mathsf{Else}\ e'\rangle}$$

$$\frac{}{\langle \rho, \mathsf{If\ T\ Then}\ e\ \mathsf{Else}\ e'\rangle \rightarrow_A \langle \rho, e\rangle} \qquad \frac{}{\langle \rho, \mathsf{If\ F\ Then}\ e\ \mathsf{Else}\ e'\rangle \rightarrow_A \langle \rho, e'\rangle}$$

Loc $$\frac{\langle \rho, e\rangle \rightarrow_A \langle \rho', e''\rangle}{\langle \rho, \mathsf{let}\ x = e\ \mathsf{in}\ e'\rangle \rightarrow_A \langle \rho', \mathsf{let}\ x = e''\ \mathsf{in}\ e'\rangle}$$

$$\frac{}{\langle \rho, \mathsf{let}\ x = v\ \mathsf{in}\ e'\rangle \rightarrow_A \langle \rho, e'[v/x]\rangle}$$

An operational semantics for *Fpl*
(the same rules in Maude)

```
rl [Var] : < rho, x > => < rho, rho(x) > .
rl [Op] : < rho, v op v' > => < rho, Ap(op,v,v') > .
crl [Op] : < rho, e op e' > => < rho', e'' op e' >
           if < rho, e > => < rho', e'' > .
crl [Op] : < rho, e op e' > => < rho', e op e'' >
           if < rho, e' > => < rho', e'' > .
crl [If] : < rho, If be Then e Else e' > =>
             < rho', If be' Then e Else e' >
           if < rho, be > => < rho', be' > .
rl [If] : < rho, If T Then e Else e' > => < rho, e > .
rl [If] : < rho, If F Then e Else e' > => < rho, e' > .
crl [Loc] : < rho, let x = e in e' > =>
              < rho', let x = e'' in e' >
            if < rho, e > => < rho', e'' > .
rl [Loc] : < rho, let x = v in e' > => < rho, e'[v / x] > .
```

A stack machine for *Fpl*

- We define another operational semantics for the functional language *Fpl*, based on an abstract stack machine.
- A state of the stack machine is a triple

```
< ST, rho, e >
```

  where

  - `ST` is a stack of values,
  - `rho` is an environment assigning values to variables, and
  - `e` is an expression.

- An initial state is a triple `< empty, rho, e >`
- A final state is a triple `< v, rho, empty >`

Analysis rules for the stack machine

```
rl [Opm1] : < ST, rho, e op e' . C > =>
            < ST, rho, e . e' . op . C > .
rl [Opm1] : < ST, rho, be op be' . C > =>
            < ST, rho, be . be' . bop . C > .
rl [Ifm1] : < ST, rho, If be Then e Else e' . C > =>
            < ST, rho, be . if(e, e') . C > .
rl [Locm1] : < ST, rho, let x = e in e' . C > =>
             < ST, rho, e . < x, e' > . C > .
rl [Notm1] : < ST, rho, Not be . C > =>
            < ST, rho, be . not . C > .
rl [Eqm1] : < ST, rho, Equal(e, e') . C > =>
            < ST, rho, e . e' . equal . C > .
```

Application rules for the stack machine

(only some of them)

```
rl [Opm2] : < v’ . v . ST, rho, op . C > =>
            < Ap(op,v,v’) . ST, rho, C > .
crl [Varm] : < ST, rho, x . C > => < v . ST, rho, C >
             if v := lookup(rho,x) .
rl [Valm] : < ST, rho, v . C > => < v . ST, rho, C > .
rl [Notm2] : < T . ST, rho, not . C > => < F . ST, rho, C > .
rl [Notm2] : < F . ST, rho, not . C > => < T . ST, rho, C > .
crl [Eqm2] : < v . v’ . ST, rho, equal . C > =>
             < T . ST, rho, C > if v = v’ .
crl [Eqm2] : < v . v’ . ST, rho, equal . C > =>
             < F . ST, rho, C > if v =/= v’ .
rl [Ifm2] : < T . ST, rho, if(e, e’) . C > =>
             < ST, rho, e . C > .
rl [Locm2] : < v . ST, rho, < x, e > . C > =>
             < ST, (x,v) . rho, e . pop . C > .
rl [Pop] : < ST, (x,v) . rho, pop . C > => < ST, rho, C > .
```

Relating both semantics for *Fpl*

- We want to show that the stack machine *implements correctly* the previous operational semantics.
- This is a particular example of the general idea of relating an abstract system with a more concrete one.
- In both semantics that we have seen for the functional language *Fpl*, the evaluation of a given expression requires the computation of several steps or transitions.
- It seems appropriate to study the relationship between the stepwise computation of both semantics.

Relating both semantics for *Fpl*

- We have two transition systems, namely, $\mathcal{S} = (S, \rightarrow_{\mathcal{S}})$ and $\mathcal{C} = (C, \rightarrow_{\mathcal{C}})$, for the stack machine and the first operational semantics, respectively.
- In order to show that the stack machine correctly implements the first operational semantics, we will prove first that *there exists a stuttering simulation* of transition systems $h : \mathcal{S} \longrightarrow \mathcal{C}$.
- Intuitively, the state `< empty, rho, e >` in S, where `empty` denotes the empty stack of values, should be related with the state `< rho, e >` in C.

An example of related states

$$\begin{array}{lll}
\texttt{< empty, empty, 2 + 3 >} & \rightarrow_S & \texttt{< empty, empty, 2 . 3 . + >} \\
 & \rightarrow_S & \texttt{< 2, empty, 3 . + >} \\
 & \rightarrow_S & \texttt{< 3 . 2, empty, + >} \\
 & \rightarrow_S & \texttt{< 5, empty, empty >}
\end{array}$$

- All the states from the first to the fourth carry the same information, although in different positions (due to the analysis rules). Therefore, it seems appropriate to relate all of them with the same state `< empty, 2 + 3 >`.
- However, in the fifth state the information has changed (due to an application rule). Now it seems appropriate to relate this state with `< empty, 5 >`.

Definition of the relation h

- We define $h : \mathcal{S} \longrightarrow \mathcal{C}$ as follows:

$$h(a) = < \texttt{rho, e} >$$

  if a can be obtained from $< \texttt{empty, rho, e} >$ with zero or more applications of the analysis rules for the stack machine together with `Valm` and `Locm2`.
- Notice that h is a *function*, precisely because not all rules are applicable in this definition.
- Moreover, h is *partial*; indeed, it is only defined for reachable states, which form a complete substructure of $\mathcal{S}$ where h is total.

Equational definition of the relation h

```
eq [Base] : h(< empty, rho, e>) = < rho, e > .
eq [Opm1] : h(< ST, rho, e . e' . op . C >) =
            h(< ST, rho, e op e' . C >) .
eq [Opm1] : h(< ST, rho, be . be' . bop . C >) =
            h(< ST, rho, be bop be' . C >) .
eq [Ifm1] : h(< ST, rho, be . if(e, e') . C >) =
            h(< ST, rho, If be Then e Else e' . C >) .
eq [Locm1] : h(< ST, rho, e. <x, e'> . C >) =
             h(< ST, rho, let x = e in e' . C >) .
eq [Notm1] : h(< ST, rho, be . not. C >) =
             h(< ST, rho, Not be . C >) .
eq [Eqm1] : h(< ST, rho, e . e' . equal . C >) =
            h(< ST, rho, Equal(e, e') . C >) .
eq [Locm2] : h(< ST, (x, v) . rho, e . pop . C >) =
             h(< v . ST, rho, < x, e > . C >) .
ceq [Valm] : h(< v . ST, rho, C >) = h(< ST, rho, v . C >)
             if not(enabled(C)) .
ceq [Valm] : h(< bv . ST, rho, C >) = h(< ST, rho, bv . C >)
             if not(enabled(C)) .
```

Characterization of the relation h

Lemma

If $h($`< ST, rho, e . C >`$)=$ `< rho, e' >`*, then there exists a position* p *in* $\mathtt{e}'$ *such that* $\mathtt{e}'|_p = \mathtt{e}$ *and, if* `e` *is not a value it will be a subexpression that can be reduced with the rules of the first operational semantics producing* $\mathtt{e}'$ *in the next step.*

Proof.

We orient the equations defining h and proceed by induction over the number of steps used to reach `< rho, e' >`. □

The function h is a stuttering simulation

Theorem

The partial function $h : \mathcal{S} \longrightarrow \mathcal{C}$ defines a stuttering simulation of transition systems.

Proof.

We use the characterization in Manolios's theorem.
Since h is a partial function, it is enough to define a function $\mu : S \times C \longrightarrow \mathbb{N}$. Specifically, $\mu(a, c)$ is the length of the longest path beginning in a and using only analysis rules.
Assume that $a \rightarrow_{\mathcal{S}} a'$ and that $h(a) = c$.
If a' is obtained applying an analysis rule, then $h(a') = c$ and $\mu(a', c) < \mu(a, c)$.
Otherwise, we must find an element c' such that $c \rightarrow_{\mathcal{C}} c'$ and $h(a') = c'$. For this, we distinguish cases according to the applied rule. □

The simulation h is not a bisimulation

- Notice that h *is not a bisimulation*, i.e., h^{-1} is not a simulation.
- In the first operational semantics, for a given expression of the form `e op e'`, we can choose whether to evaluate `e` before `e'` or the other way around, while the stack machine always evaluates first `e`.
- For example, the transition
  `< empty, (1 + 2) + (3 + 4) >` $\rightarrow_{\mathcal{C}}$
  `< empty, (1 + 2) + 7 >`
  cannot be simulated by the stack machine.

Simulation of Kripke structures

- The simulation h can be extended to Kripke structures.
- We consider as set AP of atomic propositions the set of all possible values.
- We extend the transition systems $\mathcal{S}$ and $\mathcal{C}$ with the labelling functions:
  $L_{\mathcal{S}}(\texttt{< empty, rho, v >}) = \{\texttt{v}\}$
  $L_{\mathcal{S}}(\texttt{< v, rho, empty >}) = \{\texttt{v}\}$
  $L_{\mathcal{C}}(\texttt{< rho,v >}) = \{\texttt{v}\}$
  otherwise, both $L_{\mathcal{S}}(a)$ and $L_{\mathcal{C}}(c)$ are empty.
- Applying the reflection theorem, for all expressions `e` and environments `rho`, we have

$$\mathcal{C}, \texttt{< rho, e >} \models \mathbf{AF}\texttt{v} \Longrightarrow \mathcal{S}, \texttt{< empty, rho, e >} \models \mathbf{AF}\texttt{v}$$

- That is, $\mathcal{S}$ correctly implements $\mathcal{C}$.

Simulations as rewrite relations

- The construction of the category $\mathbf{SRWTh}_{\models}$ of algebraic stuttering *maps* is quite general ...
- ... but it restricts us to work with functions.
- To avoid this drawback, we define another category

$$\mathbf{SRelRWTh}_{\models}$$

whose objects are those of $\mathbf{SRWTh}_{\models}$:

$$(\mathcal{R}, (\Sigma', E \cup D), J)$$

Simulations as rewrite relations

A morphism $(\mathcal{R}_1, (\Sigma_1', E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma_2', E_2 \cup D_2), J_2)$ in the category $\mathbf{SRelRWTh}_{\models}$, called an *algebraic stuttering simulation*, is a pair (α, H) such that:

1. (α, H) is a stuttering simulation of Kripke structures $(\alpha, H) : \mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1} \longrightarrow \mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$.
2. There exists a rewrite theory extension $\mathcal{R}_3$ containing and protecting disjoint copies of $(\Sigma_1', E_1 \cup D_1, R_1)$ and $(\Sigma_2', E_2 \cup D_2, R_2)$ in which α can be equationally defined through an operator $\alpha : Prop_1 \longrightarrow StateForm_2$, and H is defined by rewrite rules involving an operator $H : J_1(State)_1\ J_2(State)_2 \longrightarrow Bool$ such that xHy iff $\mathcal{R}_3 \vdash H(x, y) \longrightarrow true$. Here the subscripts 1, 2 indicate the corresponding names for the disjoint copies of the kinds, and $StateForm_2$ is a new kind for representing state formulas over $Prop_2$.

Simulations as rewrite relations

The functor $\mathcal{K}$ is extended in the obvious way to the new categories:

- for objects $(\mathcal{R}, (\Sigma', E \cup D), J)$,

$$\mathcal{K}(\mathcal{R}, (\Sigma', E \cup D), J) = \mathcal{K}(\mathcal{R}, J(State))_{\Pi}$$

- for morphisms $(\alpha, h) : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$,

$$\mathcal{K}(\alpha, h) = (\alpha, h)$$

Theorem (representability)

With the above definitions, $\mathcal{K} : \mathbf{SRelRWTh}_{\models} \longrightarrow \mathbf{KSSim}$ *is surjective on objects, full, and faithful.*

A Communication Protocol Example

If a communication mechanism does not provide reliable, in-order delivery of messages, it may be necessary to generate this service using the given unreliable basis. Both the sender and the receiver keep a counter for synchronization purposes; the sender releases a message together with such number and does not send another message until it receives an acknowledgment by the receiver.

```
mod PROTOCOL is
  protecting NAT .   protecting QID .
  sorts Object Msg Config .  subsort Object Msg < Config .
  op null : -> Config .
  op __ : Config Config -> Config [assoc comm id: null] .
  sorts Elem List Contents .
  subsort Elem < Contents List .
  op empty : -> Contents .
  ops a b c : -> Elem .
  op nil : -> List .
  op _:_ : List List -> List [assoc id: nil] .
```

A Communication Protocol Example

```
op to:_(_,_) : Qid Elem Nat -> Msg .
op to:_ack_ : Qid Nat -> Msg .

op <_: SND | rec:_, sendq:_ , sendbuff:_, sendcnt:_ > :
   Qid Qid List Contents Nat -> Object .

  --- rec is the receiver, sendq is the outgoing queue,
  --- sendbuff is either empty or the current data,
  --- sendcnt is the sender sequence number

op <_: RCV | sender:_, recq:_, reccnt:_ > :
   Qid Qid List Nat -> Object .

  --- sender is the sender, recq is the incoming queue,
  --- and reccnt is the receiver sequence number

vars S R : Qid .    vars M N : Nat .
var E : Elem .      var L : List .    var C : Contents .
```

A Communication Protocol Example

```
  --- rules for the sender

 rl [produce-a] :
 < S : SND | rec: R, sendq: L, sendbuff: empty, sendcnt: N >
 =>  < S : SND | rec: R, sendq: L : a, sendbuff: a,
                 sendcnt: N + 1 > .
 rl [produce-b] :
 < S : SND | rec: R, sendq: L, sendbuff: empty, sendcnt: N >
 =>  < S : SND | rec: R, sendq: L : b, sendbuff: b,
                 sendcnt: N + 1 > .
 rl [produce-c] :
 < S : SND | rec: R, sendq: L, sendbuff: empty, sendcnt: N >
 =>  < S : SND | rec: R, sendq: L : c, sendbuff: c,
                 sendcnt: N + 1 > .
 rl [send] :
 < S : SND | rec: R, sendq: L, sendbuff: E, sendcnt: N >
 =>  < S : SND | rec: R, sendq: L, sendbuff: E,
                 sendcnt: N >  (to: R (E,N)) .
```

A Communication Protocol Example

```
rl [rec-ack] :    (to: S ack M)
< S : SND | rec: R, sendq: L, sendbuff: C, sendcnt: N >
=>  < S : SND | rec: R, sendq: L,
                sendbuff: (if N == M then empty else C fi),
                sendcnt: N > .

  --- rule for the receiver

rl [receive] :    (to: R (E,N))
< R : RCV | sender: S, recq: L, reccnt: M >
=> (if N == M + 1 then
    < R : RCV | sender: S, recq: L : E, reccnt: M + 1 >
    else
    < R : RCV | sender: S, recq: L, reccnt: M >
    fi)
   (to: S ack N) .
```

A Communication Protocol Example

```
mod PROTOCOL-FAULTY is
  including PROTOCOL .
  op <_: DSTR | sender:_, rec:_, cnt:_, cnt':_, rate:_ > :
     Qid Qid Qid Nat Nat Nat -> Object .

  var M : Msg .   vars K N N' : Nat .
  var E : Elem .   vars S R D : Qid .

rl [destroy1] :    (to: R (E,N))
< D : DTR | sender: S, rec: R, cnt: N, cnt': s(N'), rate: K >
=> < D : DTR | sender: S, rec: R, cnt: N, cnt': N', rate: K >
rl [destroy2] :    (to: R ack N)
< D : DTR | sender: S, rec: R, cnt: N, cnt': s(N'), rate: K >
=> < D : DTR | sender: S, rec: R, cnt: N, cnt': N', rate: K >
rl [limited-injury] :
< D : DTR | sender: S, rec: R, cnt: N, cnt': 0, rate: K >
=> < D : DTR | sender: S, rec: R, cnt: s(N), cnt': K, rate: K
```

A Communication Protocol Example

- To check if messages are delivered in the correct order, we define a state predicate `prefix(S,R)` that holds for a sender `S` and receiver `R` whenever the queue associated to `R` is a prefix of that associated to `S`.
- This is done, both for `PROTOCOL` and `PROTOCOL-FAULTY`, by means of the following operator and equation:

```
op prefix : Qid Qid -> Prop .
var CO : Config .

eq < S : SND | rec: R, sendq: L1 : L2, sendbuff: C,
               sendcnt: N >
   < R : RCV | sender: S, recq: L1, reccnt: M >
   CO |= prefix(S, R)
=  true .
```

A Communication Protocol Example

- The initial state

```
eq init = < 'A : SND | rec: 'B, sendq: nil,
                       sendbuff: empty, sendcnt: 0 >
          < 'B : RCV | sender: 'A, recq: nil, reccnt: 0 >
```

  should satisfy the formula $\mathbf{AG}\,\texttt{prefix}(\texttt{'A}, \texttt{'B})$.

- We define a stuttering simulation

$$H : \mathcal{K}(\texttt{PROTOCOL-FAULTY}, \texttt{Config})_\Pi \longrightarrow \mathcal{K}(\texttt{PROTOCOL}, \texttt{Config})_\Pi$$

  where Π only contains the state predicate `prefix`.

- Given configurations (states) a and b respectively in `PROTOCOL-FAULTY` and `PROTOCOL`, aHb iff:
  - b is obtained from a by removing all objects of class `DTR`, or
  - there exists a' such that $a'Hb$ and a can be obtained from a' by the rules that belong only to `PROTOCOL-FAULTY`.

A Communication Protocol Example

- We can define *H* as a rewrite relation in an admissible rewrite theory extending `PROTOCOL` and `PROTOCOL-FAULTY`.
- Kinds of states have been renamed as `Config1` and `Config2`.
- `removeD` and `messages` are auxiliary functions that, given a configuration, remove all objects of class `DTR` and return all messages in it, respectively.
- We have new operators

```
op H : Config1 Config2 -> Bool .

op undo-d1 : Qid Elem Nat -> Msg .
op undo-d2 : Qid Nat -> Msg .
op undo-injury : -> Msg .
```

A Communication Protocol Example

```
rl [destroy1-inv] :    undo-d1(R,E,N)
< D : DTR | sender: S, rec: R, cnt: N, cnt': N' >
=> < D : DTR | sender: S, rec: R, cnt: N, cnt': s(N') >
   (to: R (E,N)) .
rl [destroy2-inv] :    undo-d2(R,N)
< D : DTR | sender: S, rec: R, cnt: N, cnt': N' >
=> < D : DTR | sender: S, rec: R, cnt: N, cnt': s(N') >
   (to: R ack N) .
rl [limited-injury-inv] :    undo-injury
< D : DTR | sender: S, rec: R, cnt: s(N), cnt': K, rate: K >
=> < D : DTR | sender: S, rec: R, cnt: N, cnt': 0 > .
crl H(C, C') => true if removeD(C) = C' .
crl H(C, C') => true if M (to: R (E,N)) := messages(C') /\
             (to: R (E,N)) in messages(C) = false /\
             C undo-d1(R,E,N) => C'' /\ H(C'', C') => true .
crl H(C, C') => true if M (to: R ack N) := messages(C') /\
             (to: R ack N) in messages(C) = false /\
             C undo-d2(R,E) => C'' /\ H(C'', C') => true .
crl H(C, C') => true if C undo-injury => C'' /\
             H(C'', C') => true .
```

A Communication Protocol Example

Theorem

$H : \mathcal{K}(\texttt{PROTOCOL-FAULTY}, \texttt{Config})_\Pi \longrightarrow \mathcal{K}(\texttt{PROTOCOL}, \texttt{Config})_\Pi$ is an algebraic stuttering simulation.

Proof.

H preserves the atomic propositions, because the value of the sender's and the receiver's queues, `sendq` and `recq`, are not changed. Let R_1 be the set of rules in `PROTOCOL` and let R_2 be those added in `PROTOCOL-FAULTY`, and define $\mu(a, b)$ to be the length of the longest rewrite sequence starting at a using rules in R_2. This is well-defined because R_2 is terminating.
If aHb and $a \rightarrow^1_{R_1} a'$ then, since the `DTR` class plays no role in R_1, it is $b \rightarrow^1_{R_1} b'$ with $a'Hb'$. And if $a \rightarrow^1_{R_2} a'$, by definition of H it is $a'Hb$ and $\mu(a', b) < \mu(a, b)$. Because of rule `send` there are no deadlocks in the system and hence these two alternatives cover all possibilities. Therefore, H is a stuttering Π-simulation. $\square$

A Communication Protocol Example

- By the Reflection Theorem, the existence of H shows that if $\mathbf{AG}\,\texttt{prefix}('\texttt{A}, '\texttt{B})$ holds in `PROTOCOL` then it must also hold in `PROTOCOL-FAULTY` ...
- ...but we have not proved yet that the property holds in `PROTOCOL`.
- The paper on equational abstractions defines a finite abstraction

$$G : \mathcal{K}(\texttt{PROTOCOL}, \texttt{Config})_{\Pi} \longrightarrow \mathcal{K}(\texttt{ABS-PROTOCOL}, \texttt{Config})_{\Pi}$$

  for the case of two processes.
- Then, the fact that messages are delivered in order is model checked in `ABS-PROTOCOL`.
- By composing G with H this also proves that the same property is true in `PROTOCOL-FAULTY`.

Recursive simulations

- All the previous definitions and constructions can be specialized to being recursive.
- A transition system $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ is called *recursive* if B is a recursive set and there is a recursive function $next : B \longrightarrow \mathcal{P}_{\text{fin}}(B)$ (where $\mathcal{P}_{\text{fin}}(B)$ is the recursive set of finite subsets of B) such that $a \rightarrow_{\mathcal{B}} b$ iff $b \in next(a)$.
- A Kripke structure $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ is called *recursive* if $(B, \rightarrow_{\mathcal{B}})$ is a recursive transition system, AP is a recursive set, and the function $\hat{L}_{\mathcal{B}} : B \times AP \longrightarrow Bool$ mapping a pair (a, p) to **true** if $p \in L_{\mathcal{B}}(a)$ and to **false** otherwise, is recursive.

Recursive simulations

- Let $\mathcal{R} = (\Sigma, E \cup A, R)$ be a finitary rewrite theory. We call $\mathcal{R}$ *recursive* if:
  1. there exists a *matching algorithm modulo* the equational axioms A;
  2. the equational theory $(\Sigma, E \cup A)$ is (ground) *Church-Rosser and terminating modulo* A; and
  3. the rules R are (ground) *coherent* relative to the equations E modulo A.
- If $\mathcal{R}$ is recursive, so are $\mathcal{T}(\mathcal{R})_k$ and $\mathcal{K}(\mathcal{R}, k)_\Pi$.
- Every recursive transition systems and Kripke structure can be specified with a recursive rewrite theory.

Recursive simulations

These definitions give rise to corresponding categories:

- $\mathbf{RecSRWTh}_{\models} \subseteq \mathbf{SRWTh}_{\models}$
  - objects as in $\mathbf{SRWTh}_{\models}$, but with $\mathcal{R}$ recursive
  - morphisms as in $\mathbf{SRWTh}_{\models}$ but with h defined by Church-Rosser and terminating equations
- $\mathbf{RecKSMap} \subseteq \mathbf{KSMap}$
  - objects recursive Kripke structures
  - morphisms (α, h) with α and h recursive

The representability results remain the same:

Theorem

$\mathcal{K} : \mathbf{RecSRWTh}_{\models} \longrightarrow \mathbf{RecKSMap}$ *is surjective on objects up to isomorphism, full, and faithful.*

Recursive simulations

Analogously for simulations as rewrite relations.

Theorem

$\mathcal{K}$: $\mathbf{SRelRWTh}_{\models} \longrightarrow \mathbf{KSSim}$ is surjective on objects, full, and faithful, and $\mathcal{K}$: $\mathbf{RecSRelRWTh}_{\models} \longrightarrow \mathbf{RecKSSim}$ is surjective on objects up to isomorphism, full, and faithful. Graphically:

$$\begin{tikzcd}
\mathbf{RecSRelRWTh}_{\models} \arrow[r, hook] \arrow[d, "\mathcal{K}"] & \mathbf{SRelRWTh}_{\models} \arrow[d, "\mathcal{K}"] \\
\mathbf{RecKSSim} \arrow[r, hook] & \mathbf{KSSim}
\end{tikzcd}$$

This is the most general representability result possible for stuttering simulations. It shows that we can represent both Kripke structures and stuttering simulations in rewriting logic.

Summary

- We have presented a quite general notion of stuttering simulation that relaxes the requirements on preservation of state predicates both in not requiring identical preservation and in allowing formulas to be translated.
- We have also proved general representability results showing that both Kripke structures and their simulations can be fruitfully represented in rewriting logic.
- Different ways of representing these notions in rewriting logic, ranging from equational abstractions to algebraic stuttering simulations.