Categorial Results in Compositional Model Checking

Paulo Mateus

Diploma Thesis

Supervised by Amílcar Sernadas

June 1997

Categorial Results in Compositional Model Checking

Abstract

Herein, we give algebraic foundations for compositional model checking of Moore machines. We define composition of Moore machines and maximal closing environments as categorial constructions. Afterwards, we design a language to specify Moore machines (MSL) and define a compositional operational semantics for this language. We compare GNOME objects with MSL objects. Finally, we present techniques for checking safety properties of MSL specifications.

Acknowledgements

Special thanks to:

Professor Amílcar Sernadas, for his enthusiasm in supervising this thesis, and for awakening me for computer science.

Professor Cristina Sernadas, for the motivation that always gave me during the writing of this thesis.

Jaime, for his contribution, our talks and for his positive presence during many stages of this work.

All section 84 for the excellent working environment.

My parents and Alexandra for their patience.

This work was partially supported by the PRAXIS XXI Program and FCT, as well as by PRAXIS XXI Projects 2/2.1/MAT/262/94 SitCalc, PCEX/P/MAT/46/96 ACL plus 2/2.1/TIT/1658/95 LogComp, and ESPRIT IV Working Groups 22704 ASPIRE and 23531 FIREworks.

Contents

Chapter 1

Introduction

Model checking is the branch of applied mathematics that deals with formal verification [Wol95]. Because large systems have to be checked, there has been a recent growth of interest on compositional model checking. In compositional model checking we obtain properties of a system from the properties of its components [Lon93]. Herein, we give algebraic foundations for compositional model checking of Moore machines.

1.1 Moore Machines

The main goal of this section is to obtain a categorial characterization of composition of Moore machines. We start by obtaining an appropriate definition of morphism between Moore machines. After, we define relevant categories and obtain results in these categories concerning colimits and cocartesian lifting. By taking advantage of this results, we define composition as a categorial construction, and verify that this definition matches the classic one.

1.2 Moore Specification Language

In this section we present a based object language to specify Moore machines (MSL). We also inherit the results of the previous chapter to define a compositional operational semantics of MSL. We compare GNOME with MSL objects and try to define a compositional operational semantics of GNOME [SSR96, RS95b].

1.3 MSL model checking

In this section we present techniques for checking MSL specifications. We use the temporal logic ACTL to check safety properties of MSL specifications. We define the ACTL institution and use Kripke structures as models. We define satisfaction of a formula by a Moore machines and map Moore machines to Kripke structures taking into account this definition.

Chapter 2

Moore Machines

Moore machines are a widely known model of synchronous digital circuits [Tau82]. The main goal of this section is to provide an algebraic characterization of these machines and their composition.

A Moore machine is a state transition system, with inputs and outputs, where inputs are associated to transitions and outputs to states. The inputs are supplied by the environment.

Notice that the Moore machines considered in this thesis differ from the classic ones in two ways:

- we allow nondeterminism;
- we allow inputs and outputs to take values besides boolean ones.

2.1 Signatures

Definition 2.1 A *Moore machine signature* is a triple $\Sigma = \langle A, \lambda a \in A.D_a, A_o \rangle$ where:

- $A \in |Set|$ is the set of attributes;
- $D_a \in |Set|$ is the domain of the attribute a, where:
  - $D_a \neq \emptyset$ for each $a \in A$.
- $A_o \subseteq A$ is the set of output attributes.

We denote by A_i the set of input attributes ($A_i = A \setminus A_o$).

Definition 2.2 A *Moore machine signature morphism* $\sigma : \Sigma \to \Sigma'$ is a morphism in Set $\sigma : A \to A'$ such that:

- $\sigma(A_o) \subseteq A'_o$;
- $D_{\sigma(a)} = D_a$.

Notice that signature morphisms preserve outputs. The reason for this property shall be clear later, when we introduce composition between Moore machines.

Prop/Definition 2.3 Moore Machines signatures and their morphisms constitute the category *MSig*.

proof: The identity and the composition are the identity and composition of *Set*.

□

Definition 2.4 *Labeling map of a set over a Moore signature.*

Let:

- $\Sigma = \langle A, \lambda a \in A.D_a, A_o \rangle \in |MSig|$;
- $B \subseteq A$.

We define a labeling map of B over Σ as:

$$\lambda a \in B.l(a) \text{ where } l(a) \in D_a$$

And we denote the set of all this maps by Lbl_B^Σ.

2.2 Moore Machines

Definition 2.5 A *Moore machine* is a pair $\langle \Sigma, M \rangle$ where:

- $\Sigma \in |MSig|$;
- $M = \langle S, I, R, L \rangle$ such that:
  - $S \in |Set|$;
  - $I \subseteq S, I \neq \emptyset$;
  - $R \subseteq S \times Lbl_{A_i}^\Sigma \times S$;
  - $L : S \to Lbl_{A_o}^\Sigma$.

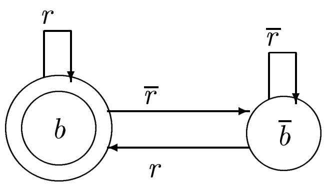
such that:

- $\forall_{s \in S} \forall_{l \in Lbl_{A_i}^\Sigma} \exists_{s' \in S} R(s, l, s')$

Moore machines are represented by a directed graph, labeled in the edges and nodes. The nodes and edges represent states and transitions. We double circled a node that denotes an initial state. Nodes are labeled with elements of $Lbl_{A_o}^\Sigma$ and edges with elements of $Lbl_{A_i}^\Sigma$.

Example 2.6 *Boolean one sized buffer:*

A boolean one sized buffer is a Moore machine with one input r and one output b. When a transition occurs, the output b takes the value received through the input r. To denote boolean values we use the following notation, b means that the value of b is true and $\overline{b}$ that is false. We consider that the initial value of b is true:



Definition 2.7 A *Moore machine morphism* is a pair $\langle \sigma, h \rangle : \langle \Sigma, M \rangle \rightarrow \langle \Sigma', M' \rangle$ where:

- $\sigma : \Sigma \rightarrow \Sigma'$ in *MSig*;
- $h : S' \rightarrow S$ in *Set*.

such that:

- $h(I') \subseteq I$;
- if $R'\langle s_1', l', s_2' \rangle$ then $R\langle h(s_1'), l, h(s_2') \rangle$
  where: $l(a) = \begin{cases} l'(\sigma(a)) & pt\ \sigma(a) \in A_i' \\ L'(s_1', \sigma(a)) & pt\ \sigma(a) \in A_o' \end{cases}$;
- $L(h(s'), a) = L'(s', \sigma(a))$ for each $a \in A_o$.

Prop/Definition 2.8 Moore Machines and their morphisms constitute the category *Moore*.

proof: We only show that the composition is the same of $Set \times Set^{op}$.
Let $\langle \sigma, h \rangle : \langle \Sigma, M \rangle \rightarrow \langle \Sigma', M' \rangle$ and $\langle \sigma', h' \rangle : \langle \Sigma', M' \rangle \rightarrow \langle \Sigma'', M'' \rangle$ be morphisms in *Moore*, then $\langle \sigma' \circ \sigma, h \circ h' \rangle : \langle \Sigma, M \rangle \rightarrow \langle \Sigma'', M'' \rangle$ is also a morphism of *Moore*.

- $h'(I'') \subseteq I'$ because $\langle \sigma', h' \rangle$ is a morphism;
  $h(h'(I'')) \subseteq h(I') \subseteq I$ because $\langle \sigma, h \rangle$ is a morphism.
- $L(h \circ h'(s''), a) = L(h'(s''), \sigma(a)) = L(s'', \sigma' \circ \sigma(a))$.
- if $R''\langle s_1'', l'', s_2'' \rangle$ then $R'\langle h'(s_1''), l', h'(s_2'') \rangle$ where:
  $l'(a) = \begin{cases} l''(\sigma'(a)) & pt\ \sigma'(a) \in A_i'' \\ L''(s_1'', \sigma'(a)) & pt\ \sigma'(a) \in A_o'' \end{cases}$ because $\langle \sigma', h' \rangle$ is a morphism,
  again we have $R\langle h \circ h'(s_1''), l, h \circ h'(s_2'') \rangle$ where:
  $l(a) = \begin{cases} l'(\sigma(a)) & pt\ \sigma(a) \in A_i' \\ L'(h'(s_1''), \sigma(a)) & pt\ \sigma(a) \in A_o' \end{cases} =$
  $\begin{cases} l''(\sigma' \circ \sigma(a)) & pt\ \sigma(a) \in A_i' \text{ and } \sigma' \circ \sigma(a) \in A_i'' \\ L''(s_1'', \sigma' \circ \sigma(a)) & pt\ \sigma(a) \in A_i' \text{ and } \sigma' \circ \sigma(a) \in A_o'' \\ L'(h'(s_1''), \sigma(a)) & pt\ \sigma(a) \in A_o' \end{cases} =$
  $\begin{cases} l''(\sigma' \circ \sigma(a)) & pt\ \sigma' \circ \sigma(a) \in A_i'' \\ L''(s_1'', \sigma' \circ \sigma(a)) & pt\ \sigma' \circ \sigma(a) \in A_o'' \end{cases}$.

$\square$

2.3 Limits

Proposition 2.9 *MSig* is finitely cocomplete.

proof: It is sufficient to obtain that there are initial objects, binary coproducts and coequalizers.

- initial objects: Consider $\langle \emptyset, \lambda a \in \emptyset.D_a, \emptyset \rangle$. It is trivially an initial object in *MSig*.

- coproducts: Let Σ_1 and Σ_2 be Moore machine signatures. The coproduct of Σ_1 with Σ_2 is :

  $$\langle \Sigma_1 \oplus \Sigma_2, i_1, i_2 \rangle$$

  where:

  - $\Sigma_1 \oplus \Sigma_2 = \langle A_1 \uplus A_2, \lambda a \in A_1 \uplus A_2.D_a, A_{1_o} \uplus A_{2_o} \rangle$

  and for each $k \in 1, 2$:

  - i_k is the inclusion in *Set*;

  finally we have to obtain that for each Moore machine signature Σ', and Moore machine signature morphisms f_1 and f_2 there is a unique morphism g that makes the following diagram to commute.

  $$\begin{tikzcd}
\Sigma_1 \arrow[r, "i_1"'] \arrow[dr, "f_1"'] & \Sigma_1 \oplus \Sigma_2 \arrow[d, dashed, "g"] & \Sigma_2 \arrow[l, "i_2"] \arrow[dl, "f_2"] \\
 & \Sigma' &
\end{tikzcd}$$

  let us choose g such that for each $k \in 1, 2$:

  - $g(i_k(a)) = f_k(a)$

  Trivially, g is a morphism and makes the diagram to commute. Furthermore if any other morphism h makes the diagram to commute it must fulfill the previous condition. Because this condition defines uniquely a morphism from $\Sigma_1 \oplus \Sigma_2$ to Σ', we have that h=g.

- coequalizers: Let f and g be Moore machine signatures morphisms from Σ_1 to Σ_2. We start to understand what happens with the attributes. Consider now the the following equivalence relation in A_2:

  $$\sim \subseteq A_2 \times A_2$$

  where:

- $b \sim b'$ provided that $\exists_{a \in A_1} f(a) = b \wedge g(a) = b'$;
- $\sim$ is the least equivalence relation that satisfies the previous condition.

It is easy to show that there is such relation, since it is the intersection of all equivalence relations that satisfy the first condition (and equivalence relations are closed to arbitrary intersections). We begin to remark that if $b \sim b'$ then $D_b = D_{b'}$. This is obvious taking into account the fact that if $f(a) = b \wedge g(a) = b'$ then $D_b = D_{b'} = D_a$ and by transitive reasoning it follows that if $b \sim b'$ then $D_b = D_{b'}$. We denote by $[b]$ the equivalence class of b.

The coequalizer of f and g is:

$$\langle \Sigma, n : \Sigma_2 \to \Sigma \rangle$$

where:

- $\Sigma = \langle A, \lambda a \in A.D_a, A_o \rangle$ and:
  * $A = A_2 / \sim$;
  * $D_{[b]} = D_b$, is well defined by the previous remark;
  * $A_o = \{[b] : b \in A_{2_o}\}$.
- $n(b) = [b]$ is the natural morphism in *Set* induced by the quotient set.

It is trivial to show that Σ is a Moore signature and n is a Moore signature morphism. We obtain now their universal property.

$$\begin{tikzcd}
\Sigma_1 \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & \Sigma_2 \arrow[r, "n"] \arrow[dr, "m"'] & \Sigma \arrow[d, "u"] \\
 & & \Sigma'
\end{tikzcd}$$

Start to notice that $n \circ f = n \circ g$ because $f(a) \sim g(a)$. Suppose that there is a m such that $m \circ f = m \circ g$, now consider $u : \Sigma \to \Sigma'$ such that $u([b]) = m(b)$:

- *good definition of* u: consider $R \subseteq A_2 \times A_2$ such that bRb' iff $m(b) = m(b')$. R is an equivalence relation, because $m \circ f = m \circ g$ it follows that $f(a)Rg(a)$ for all $a \in A_1$, thus by the definition of $\sim$, $\sim \subseteq R$. So if $b \sim b'$ then bRb', we can infer now that $m(b) = m(b')$ and so u is well defined;
- u *commutes*: $(u \circ n)(b) = u(n(b)) = u([b]) = m(b)$;
- *uniqueness of* u: suppose that v also commutes, then $(v \circ n)(b) = v([b]) = m(b)$. Thus $v = u$.

$\square$

Proposition 2.10 *Moore* has finite coproducts.

proof: It is sufficient to show that there are initial objects and binary coproducts.

- initial objects: Consider $\langle \Sigma, M \rangle$, where
  - $\Sigma=$ initial object in *MSig*;
  - $M = \langle \{\emptyset\}, \{\emptyset\}, R, L \rangle$ where:
    - $R = \{\langle \emptyset, \emptyset, \emptyset \rangle\}$;
    - $L = \{\langle \emptyset, \emptyset \rangle\}$.

  this is an object of *Moore* and is an initial one.

- coproducts: Let $\langle \Sigma_1, M_1 \rangle$ and $\langle \Sigma_2, M_2 \rangle$ be Moore machines. Their coproduct is:

$$\langle \langle \Sigma_1 \oplus \Sigma_2, M_1 \otimes M_2 \rangle, \imath_1, \imath_2 \rangle$$

  where:
  - $\Sigma_1 \oplus \Sigma_2$ is an coproduct in *MSig* ;
  - $M_1 \otimes M_2 = \langle S_1 \times S_2, I_1 \times I_2, R, L \rangle$ where:
    - $R \subseteq (S_1 \times S_2) \times Lbl_{A_i}^{\Sigma_1 \oplus \Sigma_2} \times (S_1 \times S_2)$ defined such that: $\langle \langle s_1, s_2 \rangle, f, \langle s_1', s_2' \rangle \rangle \in R$ iff $\langle s_k, f|_{A_{k_I}}, s_k' \rangle \in R_k$ for all $k \in 1..2$
    - $L : S_1 \times S_2 \to Lbl_{A_o}^{\Sigma_1 \oplus \Sigma_2}$ defined such that: $L(\langle s_1, s_2 \rangle, a) = L_k(s_k, a')$ provided that $a = i_k(a')$ with $a' \in A_{k_o}$.
  - $\imath_k = \langle i_k, \pi_k \rangle$ where $i_k \in$ *MSig* and π_k is the projection of $S_1 \times S_2$ into S_k.

  We start to remark that $\langle \Sigma_1 \oplus \Sigma_2, M_1 \otimes M_2 \rangle$ is a Moore machine. We only have to observe that for all inputs and states there is a transition to another state. This follows trivially by the fact that $\langle \Sigma_1, M_1 \rangle$ and $\langle \Sigma_2, M_2 \rangle$ also hold this statement. $\imath_1$ and $\imath_2$ are clearly Moore machine morphisms. It remains to prove the universal property of the coproduct:

  For each Moore machine $\langle \Sigma', M' \rangle$ and Moore morphisms m_1 and m_2 there is a unique morphism m that makes the following diagram to commute:

$$\begin{tikzcd}
\langle \Sigma_1, m_1 \rangle \arrow[r, "\imath_1"'] \arrow[dr, "m_1"'] & \langle \Sigma_1, m_1 \rangle \oplus \langle \Sigma_2, m_2 \rangle \arrow[d, dashed, "m"] & \langle \Sigma_2, m_2 \rangle \arrow[l, "\imath_2,"] \arrow[dl, "m_2"] \\
 & \langle \Sigma', m' \rangle &
\end{tikzcd}$$

  choose $m = \langle \sigma, h \rangle$ such that:
  - σ is the morphism obtained by the universal property of the coproduct in *MSig*
  - $h(s') = \langle h_1(s'), h_2(s') \rangle$

the hardest thing to obtain is that m is a morphism because it trivially commutes and is unique. It remains to prove that:

- $h(I') \subseteq I_1 \times I_2$. But $h(I') = h_1(I') \times h_2(I') \subseteq I_1 \times I_2$ because both m_1 and m_2 are morphisms.
- if $\langle s'_1, l', s'_2 \rangle \in R'$ then $\langle h_1(s'_1), l_{m_1}, h_1(s'_2) \rangle \in R_1$ and $\langle h_2(s'_1), l_{m_2}, h_2(s'_2) \rangle \in R_2$ where follows that $\langle h(s'_1), l_h, h(s'_2) \rangle \in R$ by the definition of R.
- $L(h(s'), a) = L_k(h_k(s'), a)$ provided that $i_k(a) = a$. $L_k(h_k(s'), a) = L(s', \sigma_k(a))$ because m_k is a morphism in *Moore*. Thus $L(h(s'), a) = L(s', \sigma(a))$.

□

Definition 2.11 A *signature output preserving morphisms* is a morphism $\sigma : \Sigma \to \Sigma'$ such that:

- $A_o \simeq \sigma(A_o) = A'_o$.

Definition 2.12 The forgetful functor $F : \mathit{Moore} \to \mathit{MSig}$.

Proposition 2.13 Cocartesian lifting for outputs preserving morphisms by F.

Let $\sigma : \Sigma \to \Sigma'$ be an output preserving morphism and $\langle \Sigma, M \rangle$ a Moore machine , a cocartesian lifting for σ and $\langle \Sigma, M \rangle$ by the forgetful functor F is:

$$\langle \sigma, id_S \rangle : \langle \Sigma, M \rangle \to \langle \Sigma', M' \rangle$$

where :

- $S' = S$;
- $I' = I$;
- $L'(s, \sigma(a)) = L(s, a)$;
- if $\langle s_1, l, s_2 \rangle \in R$ and for all $l' \in Lbl^{\Sigma'}_{A'_i}$ such that
  $l(a) = \begin{cases} l'(\sigma(a)) & pt\ \sigma(a) \in A'_i \\ L'(s_1, \sigma(a)) & pt\ \sigma(a) \in A'_o \end{cases}$
  we have that $\langle s_1, l', s_2 \rangle \in R'$
  more, if $\langle s_1, l', s_2 \rangle \in R'$ then for $l \in Lbl^{\Sigma}_{A_i}$ such that
  $l(a) = \begin{cases} l'(\sigma(a)) & pt\ \sigma(a) \in A'_i \\ L'(s_1, \sigma(a)) & pt\ \sigma(a) \in A'_o \end{cases}$
  we must have $\langle s_1, l, s_2 \rangle \in R$.

remark: Notice that if we have a $l' \in Lbl^{\Sigma'}_{A'_i}$ and a $s_1 \in S' = S$ we can define a $l \in Lbl^{\Sigma}_{A_i}$ as in the previous definition, this is used several times in this proof.

proof: We start to remark that $\langle \Sigma', M' \rangle$ is a Moore machine. L' is well defined because σ is injective for outputs. It remains to prove that R' is serial for the first two components. Let $s_1 \in S'$ and $l' \in Lbl_{A'_i}^{\Sigma'}$ we must get one $s_2 \in S'$ such that $\langle s_1, l', s_2 \rangle \in R'$. With s_1 and l' we can obtain l as in the definition of R'. Because R is serial there is s_2 such that $\langle s_1, l, s_2 \rangle \in R$ and thus $\langle s_1, l', s_2 \rangle \in R'$. By the definition is clear that $\langle \sigma, id_S \rangle$ is a Moore machine morphism.

We obtain now the universal property of the cocartesian lifting. Let $\langle \sigma'', m'' \rangle : \langle \Sigma, M \rangle \to \langle \Sigma'', M'' \rangle$ be a Moore machine morphism and let $\sigma' : \Sigma' \to \Sigma''$ be a Moore machine signature morphism such that $\sigma' \circ \sigma = \sigma''$ then there is a unique Moore machine morphism $\langle \sigma', m' \rangle : \langle \Sigma', M' \rangle \to \langle \Sigma'', M'' \rangle$ such that $\langle \sigma', m' \rangle \circ \langle \sigma, id_S \rangle = \langle \sigma'', m'' \rangle$.

$$\begin{tikzcd}
\langle \Sigma, M \rangle \arrow[rr, "\langle \sigma, id_S \rangle"] \arrow[dr, "\langle \sigma'', m'' \rangle"'] \arrow[rr, phantom, shift right=4ex, "="] & & \langle \Sigma', M' \rangle \arrow[dl, dashed, "\langle \sigma', m' \rangle"] \\
& \langle \Sigma'', M'' \rangle &
\end{tikzcd}
\qquad
\begin{tikzcd}
\Sigma \arrow[rr, "\sigma"] \arrow[dr, "\sigma''"'] \arrow[rr, phantom, shift right=4ex, "="] & & \Sigma' \arrow[dl, "\sigma'"] \\
& \Sigma'' &
\end{tikzcd}$$

$F : Moore \to MSig$

Let us choose

$$m' = m''$$

It remains to prove that $\langle \sigma', m' \rangle$ is a Moore machine morphism, because the universal property holds in this case.

- $L''(\sigma'(a'), s'') = L''(\sigma'(\sigma(a)), s'')$, σ is surjective for outputs, then $a' = \sigma(a)$ for some $a \in A_o$
  $L''(\sigma'(\sigma(a)), s'') = L''(\sigma''(a), s'')$, the diagram commutes in $MSig$
  $L''(\sigma''(a), s'') = L(a, m''(s''))$, $\langle \sigma'', m'' \rangle$ is a morphism
  $L(a, m''(s'')) = L'(\sigma(a), m''(s''))$ by definition of L'
  thus because $\sigma(a) = a'$ we have:
  $L''(\sigma'(a'), s'') = L'(a', m''(s''))$

- Let $\langle s''_1, l'', s''_2 \rangle \in R''$; because $\langle \sigma'', m'' \rangle$ is a morphism we have that $\langle m''(s''_1), l, m(s''_2) \rangle \in R$ where $l \in Lbl_{A_i}^{\Sigma}$ and
  $$l(a) = \begin{cases} l''(\sigma''(a)) & pt\ \sigma''(a) \in A''_i \\ L''(s''_1, \sigma''(a)) & pt\ \sigma''(a) \in A''_o \end{cases}$$
  consider now $l' \in Lbl_{A'_i}^{\Sigma'}$ such that
  $$l'(a) = \begin{cases} l''(\sigma'(a)) & pt\ \sigma'(a) \in A''_i \\ L''(s''_1, \sigma'(a)) & pt\ \sigma'(a) \in A''_o \end{cases}$$
  and $\bar{l} \in Lbl_{A_i}^{\Sigma}$ obtained as in the definition of R' by l' and $m''(s''_1)$:
  $$\bar{l}(a) = \begin{cases} l'(\sigma(a)) & pt\ \sigma(a) \in A'_i \\ L'(m''(s''_1), \sigma(a)) & pt\ \sigma(a) \in A'_o \end{cases}$$
  we show now that $\bar{l} = l$:

- case 1: $\sigma''(a) \in A''_i$
  $l(a) = l''(\sigma''(a))$
  because $\sigma''(a) = \sigma' \circ \sigma(a)$ and signature morphisms can not map outputs to inputs we have that $\sigma(a) \in A'_i$ and obviously we have also $\sigma'(\sigma(a)) \in A''_i$, so:
  $\bar{l}(a) = l'(\sigma(a)) = l''(\sigma'(\sigma(a)) = l''(\sigma''(a))$
  and we have that $\bar{l}(a) = l(a)$
- case 2: $\sigma''(a) \in A''_o$ and $\sigma(a) \in A'_o$
  $l(a) = L''(s''_1, \sigma''(a))$
  $\bar{l}(a) = L'(m''(s''_1), \sigma(a)) = L(m''(s''_1), a) = L''(s''_1, \sigma''(a))$ by definition of $\bar{l}$ and L', and because $\langle \sigma'', m'' \rangle$ is a morphism, so:
  $\bar{l}(a) = l(a)$
- case 3: $\sigma''(a) \in A''_o$ and $\sigma(a) \in A'_i$
  $l(a) = L''(s''_1, \sigma''(a))$
  $\bar{l}(a) = l'(\sigma(a))$; however because $\sigma''(a) \in A''_o$, we must have that $\sigma'(\sigma(a)) \in A''_o$ and so:
  $l'(\sigma(a)) = L''(s''_1, \sigma'(\sigma(a))) = L''(s''_1, \sigma''(a))$ and so
  $\bar{l}(a) = l(a)$

Notice that it is not possible to have $\sigma''(a) \in A''_i$ and $\sigma(a) \in A'_o$, because morphisms can not map outputs to inputs. l is obtained by l' and $m''(s''_1)$ as in the definition of R', thus $\langle m''(s''_1), l', m(s''_2) \rangle \in R'$. This ends the proof that $\langle \sigma', m' \rangle$ is a morphism.

$\square$

Remark 2.14 The fibre of F at Σ is denoted by $Moore_\Sigma$.

Proposition 2.15 *$Moore_\Sigma$ has initial objects.*

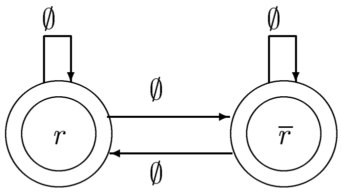
proof: Let us choose the following Moore machine $\langle \Sigma, M \rangle$ for initial object:

- $S = Lbl^{\Sigma}_{A_o}$;
- $I = S$;
- $L(l, a) = l(a)$;
- $R = S \times Lbl^{\Sigma}_{A_i} \times S$.

Let $\langle \Sigma, M' \rangle$ be a Moore machine and let us choose the following morphism: $\langle id_\Sigma, L' \rangle$ where $L' : S' \to Lbl^{\Sigma}_{A_o}$ is the output labeling function. It is trivial to verify that this is always a morphism and it is unique between $\langle \Sigma, M \rangle$ and $\langle \Sigma, M' \rangle$.

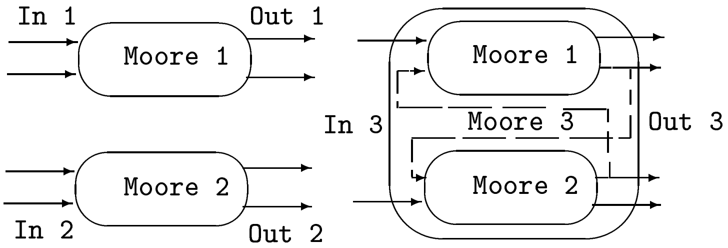
$\square$

Example 2.16 The initial object $Moore_\Sigma$ with $\Sigma = \langle \{r\}, \langle r, \{0, 1\} \rangle, \{r\} \rangle$ is:



2.4 Composition

Moore machines can be composed. The idea behind this composition is to wire some inputs of one machine to outputs of the other. The *composed* machine has all outputs of the two machines but loses the inputs that were wired.



In this section we define composition using colimits and cocartesian lifting. After, we notice that our definition matches the classic definition of composition. The first remark is that we can only wire inputs to outputs with the same domain. This wire action is denoted by a diagram. The morphisms in that diagram are called binding morphisms.

Definition 2.17 A *binding morphism* is a morphism $\sigma : \Sigma \to \Sigma'$ in *MSig* such that:

- if $a' \in A'_i$ and $\sigma(a) = a' = \sigma(b)$ then $a = b$;

remark: This property assures that two different outputs can not be wired to the same input.

With two binding morphisms, we establish a connection between outputs of one signature to inputs of another. The diagram containing the signatures that we want to compose and the binding morphisms is called composition diagram.

Definition 2.18 A *composition diagram* is a diagram $D_\Sigma^{\sigma_1,\sigma_2}$ such that:

$$D_\Sigma^{\sigma_1,\sigma_2} \equiv \Sigma_1 \xleftarrow{\sigma_1} \Sigma \xrightarrow{\sigma_2} \Sigma_2$$

such that:

- σ_i *is a binding morphism for each* $i \in \{1, 2\}$*;*
- $A_o = \emptyset$*;*
- $\sigma_1(a) \in A_{1o} \iff \sigma_2(a) \notin A_{2o}$ *for each* $a \in A$ *.*

The vertex of a colimit of a composition diagram is called the composition of Σ_1 and Σ_2 by D and is unique up to isomorphism.

Remark 2.19 If Σ_1 and Σ_2 are equal (let us say that they are equal to Σ'), the following diagrams are different[AHS90]:

$$\Sigma' \xleftarrow{\sigma_1} \Sigma \xrightarrow{\sigma_2} \Sigma' \text{ and } \Sigma \overset{\sigma_1,\sigma_2}{\rightrightarrows} \Sigma'$$

in the rightmost case we say that we have an auto-wiring situation.

Remark 2.20 We say that an input a is wired (by $D_{\Sigma}^{\sigma_1,\sigma_2}$) whenever:
$\sigma_1(a') = a$ and $\sigma_2(a') \in A''_o$ or vice versa.

Prop/Definition 2.21 Moore machine composition for auto-wiring situation.
Let:

- $\Sigma \overset{\sigma_1,\sigma_2}{\rightrightarrows} \Sigma'$ be a composition diagram D;
- $\langle \Sigma', M' \rangle$ be a Moore machine;
- $\sigma : \Sigma' \longrightarrow \Sigma''$ be the coequalizer of σ_1 and σ_2.

The auto-composition of $\langle \Sigma', M' \rangle$ by D is the codomain of the cocartesian morphism of σ and $\langle \Sigma', M' \rangle$ by F.

proof: It remains to obtain that σ is an output preserving morphism. It is clearly surjective for outputs because it is the natural map to the quotient set. It remains to obtain that it is injective for outputs ($A_o \simeq \sigma(A_o)$). Suppose that $[b] = [b']$ where $b, b' \in A'_o$ and $b \neq b'$, then by the definition of coequalizer we must have that there is an $a \in A$ such that $\sigma_1(a) = b$ and $\sigma_2(a) = b'$ however this can not be because σ_1 and σ_2 constitute a composition diagram and $b, b' \in A'_o$. $\square$

Remark 2.22 Let $\Sigma \overset{\sigma_1,\sigma_2}{\rightrightarrows} \Sigma'$ be a composition diagram for the machine $\langle \Sigma', M' \rangle$. $\langle \Sigma'', M \rangle$ is a machine obtained by this composition, where:

- Σ''=codomain of the coequalizer of σ_1 and σ_2;
- $S'' = S'$;
- $I'' = I'$;
- $L''(s, [a]) = L'(s, a)$ and it is well defined;
- $\langle s, l'', s' \rangle \in R''$ iff $\langle s, l', s' \rangle \in R'$ and we have that $l'(a) = l''([a])$ if a is not wired, and $l'(a) = L''(s, [a])$ otherwise.

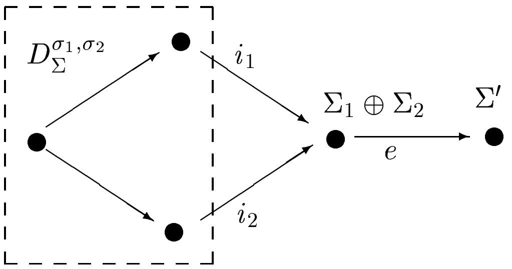
Prop/Definition 2.23 Moore machines composition (for not auto-wiring situations).
Let:

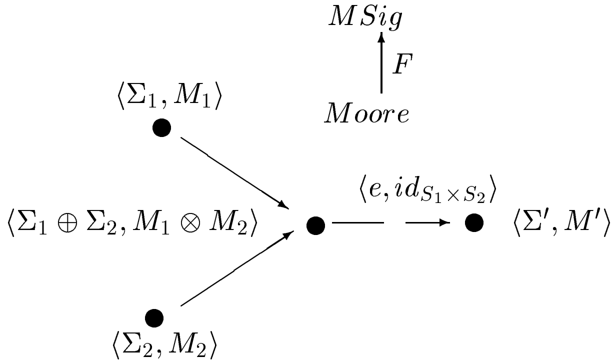
- $\langle \Sigma_1, M_1 \rangle$ and $\langle \Sigma_2, M_2 \rangle$ be Moore machines;

- $\Sigma_1 \xleftarrow{\sigma_1} \Sigma \xrightarrow{\sigma_2} \Sigma_2$ be a composition diagram D;
- σ be the coequalizer of $i_1 \circ \sigma_1$ and $i_2 \circ \sigma_2 : \Sigma \to \Sigma_1 \bigoplus \Sigma_2$.

The composition of $\langle \Sigma_1, M_1 \rangle$ and $\langle \Sigma_2, M_2 \rangle$ by D is the codomain of the cocartesian of σ and $\langle \Sigma_1, M_1 \rangle \bigoplus \langle \Sigma_2, M_2 \rangle$ by F.

proof: We must show that the coequalizer can be lifted, and afterwards we notice that the composition obtained matches the traditional composition.





Remember that the coequalizer is the natural map that maps each attribute to its equivalence class. The equivalence relation $\sim$ obtained in this case is the least such that:

- $b \sim b'$ provided that $\exists_{a \in A}\, i_1 \circ \sigma_1(a) = b \wedge i_2 \circ \sigma_2(a) = b'$.

We must show that e is bijective for outputs:

- Injective: suppose that $[o] = [o']$, $o \neq o'$ and $o, o' \in A_o^{\Sigma^1 \oplus \Sigma^2}$. In this case we must have an a such that $i_1 \circ \sigma_1(a) = o$ and $i_2 \circ \sigma_2(a) = o'$. Because i_1 and i_2 map inputs to inputs and outputs to outputs, we have that $\sigma_1(a)$ and $\sigma_2(a)$ are outputs. We have reached an absurd because σ_1 and σ_2 constitute a composition diagram.

- Surjective: this is obvious from the definition of A'_o, the quotient set of the outputs in $A_o^{\Sigma_1 \oplus \Sigma_2}$.

Thus the coequalizers obtained by composing diagrams can be lifted. Let us now understand what machine is obtained in this composition process. Consider the coproduct machine $\langle \Sigma_1 \oplus \Sigma_2, M_1 \otimes M_2 \rangle$, note that Σ' as less inputs (or equal in case we are not wiring anything) but the same outputs (because the coequalizer is an output preserving morphism). So the codomain of the cocartesian morphism is $\langle \Sigma', M' \rangle$ where:

- $S' = S_1 \times S_2$;

- $I' = I_1 \times I_2$;

- $L'(\langle s_1, s_2\rangle, [a]) = L^{\oplus}(\langle s_1, s_2\rangle, a) = L_k(s_k, a')$ provided that $a = i_k(a')$ with $a' \in A_{k_o}$;

- $\langle\langle s_1, s_2\rangle, l', \langle s_1', s_2'\rangle\rangle \in R'$ iff $\langle\langle s_1, s_2\rangle, l, \langle s_1', s_2'\rangle\rangle \in R^{\oplus}$ where l is:
$l(a) = \begin{cases} l'([a]) & pt\ [a] \in A_i' \\ L'(\langle s_1, s_2\rangle, [a]) & pt\ [a] \in A_o' \end{cases}$.

Notice that $[a] \in A_i'$ iff a is not wired to an output. So, the difference between $\langle \Sigma', M'\rangle$ and $\langle \Sigma_1 \oplus \Sigma_2, M_1 \otimes M_2\rangle$ is that we loose the wired inputs and so some transitions. The transitions that remain in the composed machine are the transitions of the coproduct machine where the values of the wired inputs match with the values of the wired outputs (in the state where the transition occurs). This is the traditional definition of Moore composition.

□

Remark 2.24 We say that a composition diagram is *multi-wiring* whenever one of the binding morphisms $\{\sigma_1, \sigma_2\}$ is not injective. In this case there are some inputs that are going to be wired to the same output. We say that we have a *simple-wiring* situation when we are neither in a auto-wiring situation nor in a multi-wiring.

Remark 2.25 Whenever we are in a simple-wiring situation it is possible to consider that the inputs and the outputs that are going to be wired have the same name in both signatures. The composition diagram consists in the intersection of attributes of each signature and the inclusions. In this case we represent the composed machine as $\langle \Sigma_1, M_1\rangle || \langle \Sigma_2, M_2\rangle$ and by $\Sigma_1 || \Sigma_2$ its signature. The composed machine satisfies the following:

- $A_1 || A_2 = A_1 \cup A_2$;

- $(A_1 || A_2)_o = A_{1_o} \cup A_{2_o}$;

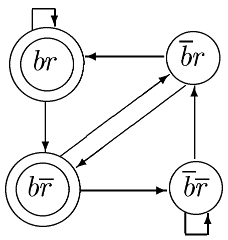
- $S_1 || S_2 = S_1 \times S_2$;

- $I_1 || I_2 = I_1 \times I_2$;

- $L_1 || L_2(\langle s_1, s_2\rangle) = L_1(s_1) \cup L_2(s_2)$;

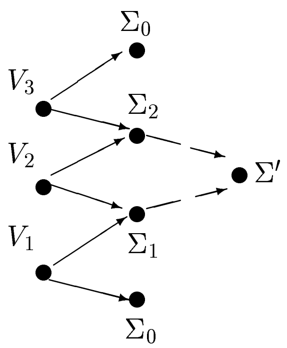
- $\langle\langle s_1, s_2\rangle, l, \langle s_1', s_2'\rangle\rangle \in R_1 || R_2$ iff $\langle s_1, l_1, s_1'\rangle \in R_1$ and $\langle s_2, l_2, s_2'\rangle \in R_2$ where $l_1(a) = l(a)$ if a is not wired and $l_1(a) = L_2(s_2, a)$ otherwise, and similar for l_2.

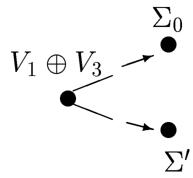
Example 2.26 The composition of the buffer (example 2.6) with the example 2.16 is represented by the following graph:



Remark 2.27 Composition of several Moore machines:

If we have three Moore machines to compose, we can compose two of them and afterwards compose the third. The technique is the following:
Let $\langle \Sigma_0, M_0 \rangle$,$\langle \Sigma_1, M_1 \rangle$ and $\langle \Sigma_2, M_2 \rangle$ be Moore machines that we would like to compose, according to the left diagram of the figure:





we could start to compose $\langle \Sigma_1, M_1 \rangle$ with $\langle \Sigma_2, M_2 \rangle$ and obtain the machine $\langle \Sigma', M' \rangle$. Remark now that there are morphisms from V_1 and V_3 to Σ' and Σ_0. So if we use the universal property of coproduct we have a unique morphism from $V_1 \oplus V_3$ to Σ' and Σ_0 that commutes with the previous. Taking into account the fact that inputs can not be wired more than once, the right diagram of the picture is still a composition diagram, thus we can proceed with the next composition. It comes from category theory that the colimit of the left diagram (without Σ' and considering that Σ_0 is only one object of the diagram) is equal to the colimit of the right diagram. Since Σ_i are arbitrary, we could have started to compose in other order that the result would be the same (because the left diagram without Σ' and where Σ_0 is only one object would be the same). We can say then that composition is associative.

2.5 Closed Moore machines

We need the concept of closed Moore machine and closing environment for model checking.

Definition 2.28 A Moore machine $\langle \Sigma, M \rangle$ is *closed* iff $A_i = \emptyset$.

Definition 2.29 *Closing environment* for a Moore machine $\langle \Sigma, M \rangle$ is a pair $\langle \langle \Sigma', M' \rangle, D \rangle$ such that:

- $D \equiv \Sigma \xleftarrow{\sigma_1} \Sigma'' \xrightarrow{\sigma_2} \Sigma'$ is a composition diagram;
- $\langle \Sigma', M' \rangle$ is a Moore machine;
- the composition of $\langle \Sigma, M \rangle$ and $\langle \Sigma', M' \rangle$ by D is closed.

Example 2.30 *The closing signature of the buffer (example 2.6) is* $\langle \{r\}, \langle r, \{0,1\}, \{r\} \rangle$.

Definition 2.31 The *closing signature* of a Moore machine signature Σ is another signature $\overline{\Sigma}$ such that:

- $\overline{A} = A_i$;
- $\lambda a \in \overline{A}.D_a = \lambda a \in A_i.D_a$;
- $\overline{A_o} = \overline{A}$.

Example 2.32 Consider the signature of the buffer (example 2.6), its closing signature is $\langle \{r\}, \langle r, \{0,1\} \rangle, \{r\} \rangle$.

Prop/Definition 2.33 The *maximal closing environment* of a Moore machine $\langle \Sigma, M \rangle$ is the closing environment $\langle \langle \overline{\Sigma}, M_I \rangle, \overline{D} \rangle$ such that:

- $\overline{D} \equiv \Sigma \xleftarrow{inc_{\overline{A}}} \langle \overline{A}, \lambda a \in \overline{A}.D_a, \emptyset \rangle \xrightarrow{id_{\overline{A}}} \overline{\Sigma}$;
- $\langle \overline{\Sigma}, M_I \rangle$ is an initial object of $Moore_{\overline{\Sigma}}$.

remark: $E(\langle \Sigma, M \rangle)$ denotes the machine $\langle \overline{\Sigma}, M_I \rangle$.
proof: We have to show that this is a closing environment, however this is trivial by the definition.

Remark 2.34 Notice that if we compose $\langle \Sigma, M \rangle$ with $E(\langle \Sigma, M \rangle)$ we are always in a simple-wiring situation, so we denote this composition by: $\langle \Sigma, M \rangle || E(\langle \Sigma, M \rangle)$. Let $\langle \Sigma', M' \rangle = \langle \Sigma, M \rangle || E(\langle \Sigma, M \rangle)$, then:

- $\Sigma' = \langle A, \lambda a \in A, A \rangle$;
- $S' = S \times Lbl^{\Sigma}_{A_i}$;
- $I' = I \times Lbl^{\Sigma}_{A_i}$;
- $L'(\langle s, l \rangle, a) = \begin{cases} l(a) & pt\ a \in A_i \\ L(s,a) & pt\ a \in A_o \end{cases}$;
- $\langle \langle s_1, l_1 \rangle, \emptyset, \langle s_2, l_2 \rangle \rangle \in R'$ iff $\langle s_1, l_1, s_2 \rangle \in R$.

Example 2.35 The maximal closing environment of the buffer (example 2.6) is the machine of example 2.16. The composition between the buffer and its maximal closing environment is the machine of example 2.26.

Chapter 3

Moore Specification Language

The syntax of MSL is inspired in GNOME[SR94, RS95a, RS95b]. It is an object based language, where object are instances of classes. The communication between objects is different from traditional communication in object oriented languages. Instead of services, a MSL object provides outputs and inputs, to give and receive information.

Although there are not any animation conditions, all objects of a MSL specification are synchronized. When one object executes an action all objects execute one action. This synchronization is due to the composition of Moore machines considered above.

3.1 MSL syntax

MSL is now introduced using BNF (Backus-Naur Form) as well as the following notation:

- normal: non terminal symbols;
- **bold**: terminal symbols;
- *italic*: non terminal symbols that are not described further.

Definition 3.1 *MSL grammar*

unity= **spec** idspec
    [*typedec...typedec*] [class...class] [object...object]
end spec

class=**class** *idclass*
    [inteface] [body] [initialization]
end *idclass*

interface=**interface** [input] [output]

input= [inputattrib..inputattrib]

inputattrib=**input** simpleattrib

output=[outputattrib...outputattrib]

outputattirb= **output** deratrib

body=**body** [stateattrib...stateattrib] [action...action]

initialization=**init** *condition*

stateattrib=**state** attrib

attrib=simpleattrib | derattrib

simpleatrib=*idatrib*:*type*

deratrib=**der** *idatrib* **from** *expression*

action=**act** *idaction* method

method=[enabling] [valuation...valuation]

enabling=**enb** *condition*

valuation=**val** *idatrib*<<*expression*

object=**object** *idobject* **instance of** *idclass*
        [bind] [initialization]
end *idobject*

bind=**bind** [wire...wire]

wire= *idinput* **to** *idobject.idoutput*

3.2 Examples of MSL specifications

We start with the specification of the one sized buffer (example 2.6).

Example 3.2 *Buffer in MSL*

```
class buffer
    interface
        input r:bool
        output der b from stvar
    body
        state stvar:bool
        act write
```

```
        val stvar<<r
    init
        stvar=true
end buffer
```

The one sized buffer in GNOME has the following specification:

Example 3.3 *Buffer in GNOME*

```
class buffer
    interface
        serv write
            par value:bool
        serv read
            res value:bool
    body
        slot buffer:bool
        serv write
            val buffer<< write.value
        serv read
            ret read.value=buffer
end buffer
```

We present a buffer with a richer interface than the previous. It only saves the input if it is different from $\perp$. This difference is important because it allows a potential user to decide when a value should be saved or not.

Example 3.4 *New-Buffer in MSL*

```
class newbuffer
    interface
        input r:bool ∪{⊥}
        output der b from stvar
    body
        state stvar:bool
        act write
            enb r≠ ⊥
            val stvar<<r
    init
        stvar=true
end newbuffer
```

Example 3.5 *stack in MSL*

```
    class stack
        interface
            input serv:enum {⊥,push,pop}
            input par:integer
            output der top from if (state=ϵ ,error,first(mystate))
            output der empty from mystate=ϵ
```

```
    body
      state mystate:list of integer
      act push
        enb serv=push
        val mystate<<prepend(mystate,par)
      act pop
        enb serv=pop and not(empty)
        val mystate<<tail(mystate)
    init
      mystate=ε
  end stack
```

We show an example where we use a stack of trees. The following list operations are used:

- first: returns the first element of the list;
- tail: returns the argument (a list) without the first element;
- prepend: returns the list that results on adding the second argument (an integer), as a prefix, to the first argument (a list).

We also use the tree operations root, right and left.

Example 3.6 *Prefix crossing of a tree*

```
spec Prefix
  class stack
    interface
      input serv:enum {⊥,push,pop}
      input serv par: tree of integer
      output der top from if (mystate=ε ,error,first(state))
      output der empty from mystate=ε
    body
      state mystate:list of tree of integer
      act push
        enb serv=push
        val mystate<<prepend(mystate,par)
      act pop
        enb serv=pop and not(empty)
        val mystate<<tail(mystate)
    init
      mystate=ε
  end stack

  class user
    interface
      input stacktop:tree of integer
      input stackempty:bool
      output der stackserv from stateserv
```

```
        output der stackpar from statepar
     body
        state mytree:tree of integer
        state seq:list of integer
        state st:enum {active,inactive,wait}
        state stateserv:enum {⊥,push,pop}
        state statepar:tree of integer
        act pushing
            enb st=active and not(empty(mytree))
            val seq<<append(seq,root(mytree))
            val mytree<<left(mytree)
            val stateserv<<push
            val statepar<<right(mytree)
            val st<<wait
        act poping
            enb st=active and not(stackempty) and empty(mytree)
            val mytree<<stacktop
            val stateserv<<pop
            val st<<wait
        act waiting
            enb st=wait
            val stateserv<< ⊥
            val st<<active
        act ending
            enb st=active and stackempty and empty(mytree)
            val st<<inactive
            val stateserv<< ⊥
     init
        st=active and empty(seq) and stateserv=⊥
  end user

  object mystack instance of stack
     bind
        serv to theuser.stackserv
        par to theuser.stackpar
     init
  end mystack

  object theuser instace of user
     bind
        stacktop to mystack.top
        stackempty to mystack.empty
     init
        mytree=T
  end theuser
end spec
```

When the state st of the object $theuser$ takes the value $inactive$, the tree **T** has been crossed and the course was saved in the state seq of the object $theuser$.

3.3 MSL semantics

We assume as given once and for all a data signature $\Sigma_{DT} = \langle S_{DT}, OP \rangle$ and an algebra dt over Σ_{DT}.

3.3.1 Semantics of a MSL object

Consider a MSL general object obj.

object obj
  interface
    input $input_1 : inptype_1$
      ...
    input $input_k : inptype_k$
    output der $output_1(outtype_1)$ from $doexpression_1$
      ...
    output der $output_m(outtype_m)$ from $doexpression_m$
  body
    state $state_1 : stype_1$
      ...
    state $state_n : stype_n$
    state der $dstate_1 : dstype_1$ by $dsexpression_1$
      ...
    state der $dstate_p : dstype_p$ by $dsexpression_p$
    act act_1
      enb enb_1
      val $st_{1,1} << expression_{1,1}$
        ...
      val $st_{1,q_1} << expression_{1,q_1}$
    ...
    act act_c
      ...
  bind
    $input_{b_1}$ to $obj_{b1}.output_{b_1}$
    ...
    $input_{b_c}$ to $obj_{b1}.output_{b_c}$
  init $init$
end obj

Definition 3.7 The *inputs* (INP^{obj}) and *outputs* (OUT^{obj}) of obj are S_{DT} indexed families of finite sets, where:

- $INP_s^{obj} = \{input_i : inptype_i = s \text{ and } i \in 1..k\}$;

- $OUT^{obj}_s = \{output_i : outtype_i = s \text{ and } i \in 1..m\}$.

buffer (example 3.2):

- $INP^{buffer}_{bool} = \{r\}$;
- $OUT^{buffer}_{bool} = \{b\}$.

Definition 3.8 The *Moore signature associated to obj* is a Moore machine signature Σ^{obj} such that:

- $A^{obj} = \bigcup_{s \in S_{DT}} INP^{obj}_s \cup OUT^{obj}_s$;
- $D_a = dt_a$;
- $A^{obj}_o = \bigcup_{s \in S_{DT}} OUT^{obj}_s$.

Definition 3.9 The *states* ($STATE^{obj}$) and *proper states* ($PSTATE^{obj}$) of *obj* are S_{DT} indexed families of finite sets, where:

- $STATE^{obj}_s = \{state_i : stype_i = s \text{ and } i \in 1..n\} \cup$ $\{dstate_i, dstype_i = s \text{ and } i \in 1..p\}$;
- $PSTATE^{obj}_s = \{state_i, stype_i = s \text{ and } i \in 1..n\}$.

buffer (example 3.2):

- $STATE^{buffer}_{bool} = \{stvar\}$;
- $PSTATE^{buffer}_{bool} = \{stvar\}$.

$DSTATE^{obj}$ denotes the family $STATE^{obj} \setminus PSTATE^{obj}$.

Definition 3.10 The *set of snapshots* (snp^{obj}) of *obj* is a set such that:

- $snp^{obj} = \bigotimes_{d \in PSTATE^{obj}} dt_d$.

We denote a projection by $p_d : snp^{obj} \to dt_d$.

buffer (example 3.2):

- $snp^{buffer} = \{0, 1\}$.

Definition 3.11 The *environment* (env^{obj}) of *obj* is a set such that:

- $env^{obj} = \bigotimes_{i \in INP^{obj}} dt_i$.

An element of env^{obj} is called *input configuration.*

buffer (example 3.2):

- $env^{buffer} = \{0, 1\}$.

Notice that $env^{obj} \simeq Lbl_{A_i}^{\Sigma^{obj}}$

Two kinds of terms occur in a MSL specification, *state* terms and *state and input* terms. Let $T_{obj}^{\Sigma_{DT}}$ denote the terms of Σ_{DT} for obj. Then $T_{obj}^{\Sigma_{DT}}(PSTATE^{obj})$ denotes the *state* terms and $T_{obj}^{\Sigma_{DT}}(PSTATE^{obj} \cup INP^{obj}) denotes the state and input terms.$ The interpretation of *state* terms depends on an algebra and a snapshot. The interpretation of *state and input* terms depends also on an input configuration. The inductive definition of the interpretation of this terms is obvious and is omitted here. We use superscript to denote interpretation.

Definition 3.12 The *Output expressions* ($expr_{out}^{obj}$) and *state expressions* ($expr_{dste}^{obj}$) of obj are maps, where:

- $expr_{out}^{obj} : OUT^{obj} \rightarrow T_{obj}^{\Sigma_{DT}}(PSTATE^{obj})$;
- $expr_{dste}^{obj} : DSTATE^{obj} \rightarrow T_{obj}^{\Sigma_{DT}}(PSTATE^{obj})$.

such that:

- $expr_{out}^{obj}(output_i) = doexpression_i,\ i \in 1..m$;
- $expr_{dste}^{obj}(dstate_i) = dsexpression_i,\ i \in 1..p$.

buffer (example 3.2):

- $expr_{out}^{buffer}(b) = stvar$.

Definition 3.13 The *set of actions* (ACT^{obj}) of obj is a set such that:

- $ACT^{obj} = \{act_1, ..., act_c\}$.

The *enabling* (enb^{obj}) and *valuation* (val^{obj}) of obj are maps where:

- $enb^{obj} : ACT^{obj} \rightarrow {T^{obj}}^{i}_{bool}$;
- $val^{obj} : ACT^{obj} \times PSTATE^{obj} \rightarrow T_{obj}^{\Sigma_{DT}}(PSTATE^{obj} \cup INP^{obj})$.

such that:

- $enb^{obj}(act_i) = enb_i$;
- $val^{obj}(act_i, d_j) = \begin{cases} expression_{i,k} & pt\ st_{i,k} = d_j \text{ for some } k \in 1..q_i \\ d_j & otherwise \end{cases}$.

buffer (example 3.2):

- $enb^{buffer}(write) = true$;
- $val^{buffer}(stvar) = r$.

Definition 3.14 The *transition* ($Trans^{obj}$) of obj is a map where:

- $Trans^{obj} : ACT^{obj} \times snp^{obj} \times env^{obj} \rightarrow snp^{obj}$.

such that:

- $p_d(Trans^{obj}(act, snap, l)) = \begin{cases} (val(act,d))^{dt,snap,l} & pt\ (enb(act))^{dt,snap,l} \\ p_d(snap) & otherwise \end{cases}$.

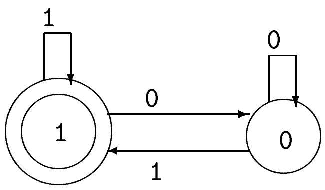
buffer (example 3.2):

- $Trans^{buffer}(write, snap, inp) = inp.$

Prop/Definition 3.15 The *Moore Machine associated* to obj is a Moore machine $\langle \Sigma^{obj}, M^{obj} \rangle$ where:

- Σ^{obj} is the Moore machine signature associated to obj;
- $M^{obj} = \langle S^{obj}, I^{obj}, R^{obj}, L^{obj} \rangle$ such that:
  - $S^{obj} = snp^{obj}$;
  - $I^{obj} = \{snap \in snp^{obj} : (init)^{dt,snap}\}$;
  - R^{obj} is such that $\langle snap, l, snap' \rangle \in R$ iff
    * $\exists_{act \in ACT} : (enb(act))^{dt,snap,l}$ and $snap' = Trans(act, snap, l)$;
    * $snap' = snap$ otherwise.
  - $L^{obj}(snap, a) = (expr^{obj}_{out}(a))^{dt,snap}$.

buffer (example 3.2):



MSL can specify any finite Moore machine.

Remark 3.16 MSL object of a (finite) Moore machine.

Let $\langle \Sigma, M \rangle$ be a (finite) Moore machine, the following MSL meta object has this Moore machine as its operational semantics:

object $\langle \Sigma, M \rangle$
  interface
    input $a_i \in A_i : D_{a_i}$
    output der $a_o \in A_o : D_{a_o}$ by $L(st, a_o)$
  body
    state st:S
    act $\langle s, l, s' \rangle$
      enb $st = s$ and $\langle st, l, s' \rangle \in R$
      val st<<s'
  init
    $s \in I$
end $\langle \Sigma, M \rangle$

3.3.2 Semantics of a MSL specification

Because composition of Moore machines is associative and commutative up to isomorphism, it is sufficient to compose two Moore machines.

object o_1
  bind
    $input^{o_1}_{b_1}$ to $obj^{o_1}_{b_1}.output^{o_1}_{b_1}$
    ...
    $input^{o_1}_{b_c}$ to $obj^{o_1}_{b_c}.output^{o_1}_{b_c}$
  init
    $init1$
end $obj1$
object o_2
  bind
    $input^{o_2}_{b_1}$ to $obj^{o_2}_{b_1}.output^{o_2}_{b_1}$
    ...
    $input^{o_2}_{b_d}$ to $obj_{b_d}.output^{o_2}_{b_d}$
  init
    $init2$
end $obj1$

Definition 3.17 The *set of wiring attributes* of two MSL objects o_1, o_2 is a set such that:

- $WIRE^{o_1,o_2} = \{input^{o_1}_{b_i} \in INP^{o_1} : obj^{o_1}_{b_i} = o_2\} \cup \{input^{o_2}_{b_i} \in INP^{o_2} : obj^{o_2}_{b_i} = o_1\}$.

Definition 3.18 The *signature composition diagram induced by* o_1,o_2 is a composition diagram D^{o_1,o_2} such that:

- $D^{o_1,o_2} = \Sigma^{o_1} \xleftarrow{\sigma_1} \Sigma \xrightarrow{\sigma_1} \Sigma^{o_2}$.

where :

- $\Sigma = \langle WIRE^{o_1,o_2}, \lambda a \in WIRE^{o_1,o_2}.dt_a, \emptyset\rangle$;
- $\sigma_1(input^{o_1}_{b_i}) = input^{o_1}_{b_i}$ for $i \in 1..c$;
- $\sigma_1(input^{o_2}_{b_i}) = output^{o_1}_{b_i}$ for $i \in 1..c$;
- $\sigma_2(input^{o_2}_{b_i}) = input^{o_2}_{b_i}$ for $i \in 1..d$;
- $\sigma_2(input^{o_1}_{b_i}) = output^{o_2}_{b_i}$ for $i \in 1..d$.

3.4 MSL vs Gnome

In this section we propose to emulate GNOME objects communication with MSL. The following restrictions are made to GNOME:

- no classes, classes are not types;
- services are called by a finite number of objects;
- each object has one service that can emulate many by taking advantage of the parameter and the enabling conditions of alternatives;
- one slot, one argument and one return (we can always consider this);
- static objects (no births and no deaths).

Definition 3.19 *Gnome general object:*

object xpto
  interface
    serv *serv*
      par *par*:int
      res *res*:int
  body
    slot *slot*:int
    serv *serv*
      alt s_1
        enb enb_{s1}
        call $serv_{s11}$ of $object_{s11}$
          arg arg_{s11}=val_{s11}
        ...
        call $serv_{s1m_1}$ of $object_{s1m_1}$
          arg arg_{s1m_1}=val_{s1m_1}
        val slot $<<$ $valslot_{s1}$
        ret $serv.ret = valret_{s1}$
      ...
      alt s_k
        enb enb_{sk}
        call $serv_{sk1}$ of $object_{sk1}$
          arg arg_{sk1}=val_{sk1}
        ...
        call $serv_{skm_k}$ of $object_{skm_k}$
          arg arg_{skm_k}=val_{skm_k}
        val slot $<<$ $valslot_{sk}$
        ret $serv.ret = valret_{sk}$
    act *act*
      alt a_1
        enb enb_{a1}
        ...

alt a_l

...

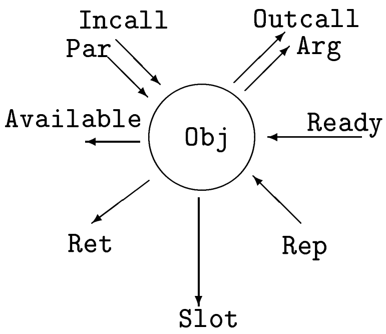
end xpto

Let *incallxpto* be the maximum number of objects that call the service of xpto. Let *outcallxpto* be the number of objects that xpto call. Remark that the service of xpto has k alternatives and the action has l alternatives.

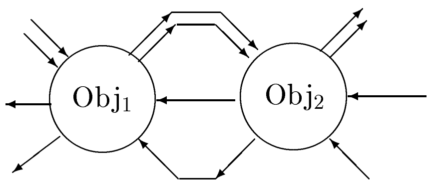
3.4.1 The interface

We need the following inputs and outputs to emulate the service call process, the action and the slot.

input Incall:array of enum {ask,fire,dismiss,free} size *incallxpto*
input Par:array of int size *incallxpto*
input Ready:array of enum {yes,no,wait,done} size *outcallxpto*
input Rep:array of int size *outcallxpto*
output der Outcall from stOutcall
output der Arg from stArg
output der Available from if(status=expecting,Available[1..*incallxpto*]=no ,stAvailable)
output der Ret from stRet
output der Slot: from stSlot



The binding conditions are clear from the following picture:



3.4.2 The body

state stArg:array of int size *outcallxpto*
state stAvailable:array of enum {yes,no,wait,done} size *incallxpto*

state stRet:int
state stSlot:int
state status: enum {expecting,$waitenb$,$waitenba$,$waitenb_{s1}$,...,
$waitenb_{sk}$,$waitfire_{s1}$,...,$waitdone_{s1}$,...,
$waitfree_{s1}$,...,$waitenb_{a1}$,...,$waitenb_{al}$,$waitdone_{a1}$,...$waitdone_{al}$}
state backupar:int
state backobject:int
state servalt: array of bool size k
state actalt: array of bool size l

We start to emulate the service call. When an object waits for a call or is not executing an action its **status** takes the value *expecting*. In GNOME, an object can only answer a call at a time. If only one object is asking for the service, *xpto* changes the **status** to *waitenb*, and communicates to the other object to *wait* until enabling is determined. Finally *xpto* backups the id of the calling object and the parameter of the call:

act *validcall*
  enb status=expecting and $\exists^1_{o \in 1..incallxpto}$ Incall[o]=ask
  val status<< *waitenb*
  val stAvailable[o] <<wait
  val stAvailable[1..o-1,o+1..incallxpto] <<no
  val servalt[i] << enb_{si} $i \in 1..k$
  val backobject<< o
  val backupar<<Par[o]

If no alternative is enabled then the service is not available:

act *notavailable*
  enb status=*waitenb* and $servalt[1..k] = 0$
  val status<<expecting

For an alternative to be enabled two things must happen, the enabling condition must be true, and all calls for external objects (outcalls) must be possible. *xpto* waits for objects to communicate if the calls are possible or not. For instance for the first alternative we have:

act ask_{s1}
  enb status=*waitenb* and $servalt[1]$
  val status<< $waitenb_{s1}$
  val stOutcall[o_{s1j}] <<ask (for all outcalls of this alt)
  val stArg[o_{s1j}] << val_{s1j}

Two things can happen, or one or more calls are not ready or everyone is. It is possible that for some objects the call is available and for others don't. If one object communicates *no* through the input **Ready** *xpto dismiss* those that have answered *yes* through the output **Outcall**.

act $notenb_{s1}$

enb status=$waitenb_{s1}$ and $\exists_{o_{s1j}}$ Ready[o_{s1j}]=no and $\forall_{o_{s1p}}$ Ready[o_{s1p}]=yes or Ready[o_{s1p}]=no

val status<< $waitenb$

val servalt[1] << 0

val stOutcall[o_{s1p}] <<dismiss (for all outcalls that said yes)

By other hand, if every call is available, then the alternative is available, and so is the service. However the object that has called *xpto* might have called other objects that were not available. So *xpto* waits for its communication informing if it can *fire* the alternative or if it is *dismiss*:

act $available_{s1}$

enb status=$waitenb_{s1}$ and $\forall_{o_{s1p}}$ Ready[o_{s1p}]=yes

val status<< $waitfire_{s1}$

val stAvailable[backobject]<<yes

Once more, two things can happen. Or the call for the service is dismissed, or is allowed to go.

act $dismiss_{s1}$

enb status=$waitfire_{s1}$ and Incall[backobject]=dismiss

val status<<expecting

val stOutcall[o_{s1p}] <<dismiss (for all outcall objects)

act $fire_{s1}$

enb status=$waitfire_{s1}$ and Incall[backobject]=fire

val status<< $waitdone_{s1}$

val stOutcall[o_{s1p}] <<fire

When all *outcalled* objects have execute their services, they send this information by the **Ready** input (that take the value *done*), and then *xpto* must *free* these objects. *xpto* receive from the input **Rep** the replies to its outcalls, and then it may execute the service.

act $done_{s1}$

enb status=$waitdone_{s1}$ and $\forall_{o_{s1p}}$ Ready[o_{s1p}]=done

val status<< $waitfree_{s1}$

val stSlot<< $valslot_{s1}$

val stRet<< $valret_{s1}$

val stOutcall[o_{s1p}] <<free

val stAvailable[backobject]<<done

Finally *xpto* only has to wait for its caller to free him:

act $free_{s1}$

enb status=$waitfree_{s1}$ and Incall[backobject]=free val status<<expecting

The action is easier to emulate because there isn't a caller. So the interaction issues are only about outcall objects. The bottom action is also emulated (since MSL does not have it), and is enable whenever the **status** is *expecting*.

Definition 3.20 Translation to MSL of the GNOME general object

object Moorexpto
  interface
    input Incall:array of enum {ask,fire,dismiss,free} size *incallxpto*
    input Par:array of int size *incallxpto*
    input Ready:array of enum {yes,no,wait,done} size *outcallxpto*
    input Rep:array of int size *outcallxpto*
    output der Outcall from stOutcall
    output der Arg from stArg
    output der Available from if(status=expecting,Available[1..*incallxpto*]=no ,stAvailable)
    output der Ret from stRet
    output der Slot from stSlot
  body
    state stArg:array of int size *outcallxpto*
    state stAvailable:array of enum {yes,no,wait,done} size *incallxpto*
    state stRet:int
    state stSlot:int
    state status: enum {expecting,$waitenb$,$waitenba$,$waitenb_{s1}$,...,
        $waitenb_{sk}$,$waitfire_{s1}$,...,$waitdone_{s1}$,...,
        $waitfree_{s1}$,...,$waitenb_{a1}$,...,$waitenb_{al}$,$waitdone_{a1}$,...}
    state backupar:int
    state backobject:int
    state servalt: array of bool size k
    state actalt: array of bool size l
    act *validcall*
      enb status=expecting and $\exists^1_{o \in 1..incallxpto}$ Incall[o]=ask
      val status<< *waitenb*
      val stAvailable[o] <<wait
      val stAvailable[1..o-1,o+1..incallxpto] <<no
      val servalt[i] << enb_{si} $i \in 1..k$
      val backobject<< o
      val backupar<<Par[o]
    act *notavailable*
      enb status=*waitenb* and $servalt[1..k] = 0$
      val status<<expecting
    act ask_{s1}
      enb status=*waitenb* and $servalt[1]$
      val status<< $waitenb_{s1}$
      val stOutcall[o_{s1j}] <<ask (for all outcallsof this alt)
      val stArg[o_{s1j}] << val_{s1j}

act $notenb_{s1}$
enb status=$waitenb_{s1}$ and $\exists_{o_{s1j}}$ Ready[o_{s1j}]=no and
$\forall_{o_{s1p}}$ Ready[o_{s1p}]=yes or Ready[o_{s1p}]=no
val status<< $waitenb$
val servalt[1] << 0
val stOutcall[o_{s1p}] <<dismiss (for all outcalls that said yes)
act $available_{s1}$
enb status=$waitenb_{s1}$ and $\forall_{o_{s1p}}$ Ready[o_{s1p}]=yes
val status<< $waitfire_{s1}$
val stAvailable[backobject]<<yes
act $dismiss_{s1}$
enb status=$waitfire_{s1}$ and Incall[backobject]=dismiss
val status<<expecting
val stOutcall[o_{s1p}] <<dismiss (for all outcall objects)
act $fire_{s1}$
enb status=$waitfire_{s1}$ and Incall[backobject]=fire
val status<< $waitdone_{s1}$
val stOutcall[o_{s1p}] <<fire
act $done_{s1}$
enb status=$waitdone_{s1}$ and $\forall_{o_{s1p}}$ Ready[o_{s1p}]=done
val status<< $waitfree_{s1}$
val stSlot<< $valslot_{s1}$
val stRet<< $valret_{s1}$
val stOutcall[o_{s1p}] <<free
val stAvailable[backobject]<<done
act $free_{s1}$
enb status=$waitfree_{s1}$ and Incall[backobject]=free
val status<<expecting enb status=$waitenb$ and $servalt[k]$
val status<< $waitenb_{sk}$
val stOutcall[o_{skj}] <<ask (for all outcallsof this alt)
val stArg[o_{skj}] << val_{skj}
act $notenb_{sk}$
enb status=$waitenb_{sk}$ and $\exists_{o_{skj}}$ Ready[o_{skj}]=no and
$\forall_{o_{skp}}$ Ready[o_{skp}]=yes or Ready[o_{skp}]=no
val status<< $waitenb$
val servalt[k] << 0
val stOutcall[o_{skp}] <<dismiss (for all outcalls that said yes)
act $available_{sk}$
enb status=$waitenb_{sk}$ and $\forall_{o_{skp}}$ Ready[o_{skp}]=yes
val status<< $waitfire_{sk}$
val stAvailable[backobject]<<yes
act $dismiss_{sk}$
enb status=$waitfire_{sk}$ and Incall[backobject]=dismiss
val status<<expecting
val stOutcall[o_{skp}] <<dismiss (for all outcall objects)
act $fire_{sk}$
enb status=$waitfire_{sk}$ and Incall[backobject]=fire

val status<< $waitdone_{sk}$
val stOutcall[o_{skp}] <<fire
act $done_{sk}$
enb status=$waitdone_{sk}$ and $\forall_{o_{skp}}$ Ready[o_{skp}]=done
val status<< $waitfree_{sk}$
val stSlot<< $valslot_{sk}$
val stRet<< $valret_{sk}$
val stOutcall[o_{skp}] <<free
val stAvailable[backobject]<<done
act $free_{sk}$
enb status=$waitfree_{sk}$ and Incall[backobject]=free
val status<<expecting
act $actionstart$
enb status=expecting
val status<< $waitenba$
val stAvailable[1...incallxpto] <<no
val actalt[i] << enb_{ai} $i \in 1..l$
act $notavailable$
enb status=$waitenba$ and $actalt[1..l] = 0$
val status<<expecting
act ask_{a1}
enb status=$waitenba$ and $actalt[1]$
val status<< $waitenb_{a1}$
val stOutcall[o_{a1j}] <<ask (for all outcallsof this alt)
val stArg[o_{a1j}] << val_{a1j}
act $notenb_{a1}$
enb status=$waitenb_{a1}$ and $\exists_{o_{a1j}}$ Ready[o_{a1j}]=no and
$\forall_{o_{a1p}}$ Ready[o_{a1p}]=yes or Ready[o_{a1p}]=no
val status<< $waitenba$
val actalt[1] << 0
val stOutcall[o_{a1p}] <<dismiss (for all outcalls that said yes)
act $available_{a1}$
enb status=$waitenb_{a1}$ and $\forall_{o_{a1p}}$ Ready[o_{a1p}]=yes
val status<< $waitdone_{a1}$
val Outcall[o_{a1p}] <<fire
act $done_{a1}$
enb status=$waitdone_{a1}$ and $\forall_{o_{a1p}}$ Ready[o_{a1p}]=done
val status<<expecting
val stSlot<< $valslot_{a1}$
val stOutcall[o_{a1p}] <<free
...
act ask_{al}
enb status=$waitenba$ and $actalt[l]$
val status<< $waitenb_{al}$
val stOutcall[o_{alj}] <<ask (for all outcallsof this alt)
val stArg[o_{alj}] << val_{alj}
act $notenb_{al}$

enb status=$waitenb_{al}$ and $\exists_{o_{alj}}$ Ready[o_{alj}]=no and
$\forall_{o_{alp}}$ Ready[o_{alp}]=yes or Ready[o_{alp}]=no
val status<< $waitenba$
val actalt[l] << 0
val stOutcall[o_{alp}] <<dismiss (for all outcalls that said yes)
act $available_{al}$
enb status=$waitenb_{al}$ and $\forall_{o_{alp}}$ Ready[o_{alp}]=yes
val status<< $waitdone_{al}$
val Outcall[o_{alp}] <<fire
act $done_{al}$
enb status=$waitdone_{al}$ and $\forall_{o_{alp}}$ Ready[o_{alp}]=done
val status<<expecting
val stSlot<< $valslot_{al}$
val stOutcall[o_{alp}] <<free
act bottom
enb status=expecting
bind
Incall[p] to obj_p.Outcall[o_p]
Par[p] to obj_p.Arg[o_p]
Ready[o_{xim_i}] to $object_{xim_i}$.Available[p_{xim_i}]
Rep[o_{xim_i}] to $object_{xim_i}$.Ret
init
status<<expecting
end Moorexpto

Remark 3.21 The results obtained here can be continued to provide a compositional operational semantics to communication between GNOME objects. The steps to take are the following:

- Define a GNOME machine based on Moore machines with hiding actions that handle communication;
- Obtain communication between GNOME machines by lifting Moore machine results presented in chapter 1.

Chapter 4

Model Checking in MSL

In this chapter we present techniques to check properties of MSL specifications. MSL has an operational semantics based in Moore machines and their composition, so it is sufficient to check properties of Moore machines.

4.1 Kripke Structures and ACTL

To represent the logic model, we take advantage of institutions[GB84, GB92].

We use a branching temporal logic, ACTL, for writing our properties. We can express safety properties with ACTL. Herein, ACTL formulas are extended from the traditional ones to fit our model checking.

Definition 4.1 A *ACTL signature* is a pair $\Pi = \langle C, \lambda c \in C.D_c \rangle$ such that:

- $C \in |Set|$;
- $D_c \in |Set|$, $D_c \neq \emptyset$ for each $c \in C$.

Definition 4.2 A *ACTL signature morphism* $\pi : \Pi_1 \to \Pi_2$ is a morphism in *Set* $\pi : C_1 \to C_2$ such that:

- $D_{\pi(c)} = D_c$.

Prop/Definition 4.3 ACTL signatures and their morphisms constitute the category KSig.

Definition 4.4 Functor $Sen : KSig \to Set$.
Let $\Phi = Sen(\langle C, \lambda c \in C.D_c \rangle)$, then:

- $true \in \Phi$;
- $(c = d) \in \Phi$ provided that $c \in C$ and $d \in D_c$;
- $\neg\, \phi \in \Phi$ provided that ϕ is atomic;
- $\phi \wedge \psi, \phi \vee \psi \in \Phi$ provided that $\phi, \psi \in \Phi$

- $\mathsf{A}\,\mathsf{X}\,\phi$, $\mathsf{A}\,\phi\,\mathsf{U}\,\psi$ $\mathsf{A}\,\phi\,\mathsf{V}\,\psi$ $\in \Phi$ provided that $\phi, \psi \in \Phi$.

$Sen(\pi)$ preserves the structure of the formula.

Kripke structures differ from the traditional because they have a set of initial worlds, Y, and the valuations deal with values besides boolean ones.

Definition 4.5 A *Kripke structure* for a ACTL signature Π is a tuple $K_\Pi = \langle W, Y, Z, V \rangle$ where:

- $W \in |Set|$;
- $Y \subseteq W, Y \neq \emptyset$;
- $Z \subseteq W \times W$;
- $V = \lambda w \in W \lambda c \in C.V(w, c) : V(w, c) \in D_c$.

Remark 4.6 When the signature is known, we represent a Kripke structure only by K.

Definition 4.7 Functor $Mod : KSig \to Set^{op}$ where :

- $Mod(\Pi)$ is the set of all Kripke structures with signature Π;
- $Mod(\pi : \Pi_1 \to \Pi_2)(\langle W, Y, Z, V \rangle) = \langle W, Y, Z, V' \rangle$. where:
  - $V'(w, a) = V(w, \pi(a))$.

Definition 4.8 Path over a Kripke structure K

$$\alpha = \alpha_0 \alpha_1 ... \alpha_n ...$$

where:

- $\alpha_i \in W$ for each $i \in I\!N$;
- $\langle \alpha_i, \alpha_{i+1} \rangle \in Z$ for each $i \in I\!N$.

we denote the set of all this paths by $Path_K$.

we denote the set of all paths such that $\alpha_0 = w$ by $Path^w_K$.

Definition 4.9 *Satisfaction of a ACTL formula*

Let K_Π be a Kripke structure, we define:

$$K, w \Vdash_\Pi \phi \text{ where } \phi \in Sen(\Pi)$$

by:

- $K, w \Vdash_\Pi true$;
- $K, w \Vdash_\Pi (c = d)$ iff $V(w, c) = d$;

- $K, w \Vdash_\Pi \neg\, \phi$ if is not the case $K, w \Vdash_\Pi \phi$;
- $K, w \Vdash_\Pi \phi \wedge \psi$ iff $K, w \Vdash_\Pi \phi$ and $K, w \Vdash_\Pi \psi$;
- $K, w \Vdash_\Pi \phi \vee \psi$ iff $K, w \Vdash_\Pi \phi$ or $K, w \Vdash_\Pi \psi$;
- $K, w \Vdash_\Pi \mathsf{A}\,\mathsf{X}\,\phi$ iff for each $\alpha \in Path^w_K$ $K, \alpha_1 \Vdash_\Pi \phi$;
- $K, w \Vdash_\Pi \mathsf{A}\,\phi\,\mathsf{U}\,\psi$ iff for each $\alpha \in Path^w_K$ there exists j such that $K, \alpha_j \Vdash_\Pi \psi$ and for all $i < j$ $K, \alpha_i \Vdash_\Pi \phi$
- $K, w \Vdash_\Pi \mathsf{A}\,\phi\,\mathsf{V}\,\psi$ iff for each $\alpha \in Path^w_K$ for all j and if for all i<j we have $K, \alpha_i \nVdash_\Pi \phi$ then $K, \alpha_j \Vdash_\Pi \psi$.

we define also:

$$K \Vdash_\Pi \phi \text{ if for every } w \in Y\ K, w \Vdash_\Pi \phi$$

Prop/Definition 4.10 The ACTL institution $\langle KSig, Sen, Mod, \Vdash\rangle$.

proof: It is trivial, by structural induction in the formulae, to obtain the satisfaction condition .

4.2 Moore to Kripke

In Moore machine we check properties about inputs and outputs. Let Σ be a Moore machine signature, the properties we can check on a Moore machine with this signature must be expressed by $Sen(\Pi_\Sigma)$ where $\Pi_\Sigma = \langle A, \lambda a \in A, D_a\rangle$. Notice that if a Moore machine is closed it is almost a Kripke structure.

Definition 4.11 Satisfaction of a formula by a closed Moore Machine

$$\langle\Sigma, M\rangle \Vdash \phi \text{ with } \phi \in Sen(\Pi_\Sigma)$$

whenever:

- $\langle\Sigma, M\rangle$ is closed ;
- $\langle S, I, Z, L\rangle \Vdash_{\Pi_\Sigma} \phi$ where:
  - $\langle s, s'\rangle \in Z$ iff $\langle s, \emptyset, s'\rangle \in R$.

For not closed Moore machines we have:

Definition 4.12 *Satisfaction of a formula by a Moore Machine*

$$\langle\Sigma, M\rangle \Vdash \phi \text{ with } \phi \in Sen(\Pi)$$

whenever:

- for all $\langle\langle\Sigma', M'\rangle, D\rangle$ closing environments of $\langle\Sigma, M\rangle$;
- the composition of $\langle\Sigma, M\rangle$ and $\langle\Sigma', M'\rangle$ by $D \equiv \langle\Sigma'', M''\rangle$;
- we have that $\langle\Sigma'', M''\rangle \Vdash \phi$.

For ACTL we can obtain the following result:

Proposition 4.13 Let $\langle \Sigma, M \rangle$ be a Moore machine, then:

$$\langle \Sigma, M \rangle \Vdash \phi \text{ iff } E(\langle \Sigma, M \rangle) || \langle \Sigma, M \rangle \Vdash \phi$$

proof: We start to show that if $E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle \Vdash \phi$ then $\langle \Sigma, M \rangle \Vdash \phi$ Let $\langle \Sigma'', M'' \rangle$ be a machine obtained by a closing environment $\langle \langle \Sigma', M' \rangle, D \rangle$. Notice that all inputs of Σ are wired to some outputs of Σ'. So to each state in S' we can associate an $l^{s'} \in Lbl^{\Sigma}_{A_i}$ where $l^{s'}(a) = L'(s', a')$ and a is wired to a'. Let us map $S'' = S \times S'$ to $S \times Lbl^{\Sigma}_{A_i}$ by the following way:

$$f(\langle s, s' \rangle) = \langle s, l^{s'} \rangle$$

Remark that if $\langle s, s' \rangle \in I''$ then $\langle s, l^{s'} \rangle$ is also an initial state in $E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle$ because $s \in I$. Notice also that if $\langle \langle s_1, s'_1 \rangle, \emptyset, \langle s_2, s'_2 \rangle \rangle \in R''$ then $\langle s_1, l^{s'}, s_2 \rangle \in R$ and so $\langle f(\langle s_1, s'_1 \rangle), \emptyset, f(\langle s_2, s'_2 \rangle) \rangle \in R^{E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle)}$. Using this two last results is easy to prove by structural induction in the ACTL formula ϕ that (we are representing the Kripke structures by closed Moore Machines):

$$\text{if } E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle, f(\langle s, s' \rangle) \Vdash \phi \text{ then } \langle \Sigma'', M'' \rangle, \langle s, s' \rangle \Vdash \phi$$

Suppose that $\langle \Sigma'', M'' \rangle \nVdash \phi$ then there is an initial state such that $\langle \Sigma'', M'' \rangle, \langle s, s' \rangle \nVdash \phi$ and so by the previous result (the contrapositive) we have that $E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle), f(\langle s, s' \rangle) \nVdash \phi$ and because $f(\langle s, s' \rangle)$ is an initial state we have that $E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle \nVdash \phi$. Thus we have shown that:
if $\langle \Sigma'', M'' \rangle \nVdash \phi$ then $E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle \nVdash \phi$
and so if $E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle \Vdash \phi$ then $\langle \Sigma'', M'' \rangle \Vdash \phi$. Because we have taken an arbitrary closing environment we obtain the following:
if $E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle \Vdash \phi$ then $\langle \Sigma, M \rangle \Vdash \phi$

It remains to show that if $\langle \Sigma', M' \rangle \Vdash \phi$ then $E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle \Vdash \phi$, however this is trivial by contrapositive. Assume that $E(\langle \Sigma, M \rangle)||\langle \Sigma, M \rangle \nVdash \phi$, then there is a closing environment that does not satisfy ϕ and so $\langle \Sigma, M \rangle \nVdash \phi$.

$\square$

This results allow us to map all Moore machines (and not only the closed ones) to Kripke structures.

Definition 4.14 The map $k : |Moore| \to Kripke$, such that:

$$k(\langle \Sigma, M \rangle) = K_{\Pi}$$

where :

- $\Pi = \langle A, \lambda a \in A.D_a \rangle$;
- $K = \langle W, Y, Z, V \rangle$ where :
  - $W = S \times Lbl^{\Sigma}_{A_i}$;

- $Y = I \times Lbl^{\Sigma}_{A_i}$;
- $Z(\langle s_0, l_0\rangle, \langle s_1, l_1\rangle)$ iff $R(s_0, l_0, s_1)$;
- $V(\langle s, l\rangle, a) = \begin{cases} l(a) & \text{iff } a \in A_i \\ L(s,a) & \text{otherwise} \end{cases}$.

We obtain then trivially:

Corollary 4.15 Satisfaction of ACTL formulae:

$$\langle \Sigma, M\rangle \Vdash \phi \text{ iff } k(\langle \Sigma, M\rangle) \Vdash_{\Pi} \phi$$

where $\phi \in Sen(\Pi)$

proof: It is trivial from proposition 4.13 by noticing that
$k(\langle \Sigma, M\rangle) = E(\langle \Sigma, M\rangle)||\langle \Sigma, M\rangle)$ (we are considering closed machines as Kripke structures).

$\square$

Chapter 5

Conclusions

We started by finding an appropriate definition of morphism for Moore machines. After, we defined relevant categories and presented results about these categories. We defined composition of Moore machines and maximal closing environment and provided categorial characterization of these constructions. We also confirmed that the definitions obtained here match with the classic ones.

We designed an object based language to specify Moore machines, MSL, and defined a compositional operational semantics for MSL specifications. In this semantics, we associated MSL objects with Moore machines and we interpreted communication between objects as composition of Moore machines. We gave a step in defining a compositional operational semantics for GNOME, but did not present a categorial characterization of GNOME communication.

We presented an ACTL institution that fitted our interests and presented techniques for compositional model checking of MSL specifications. In particular, we mapped Moore machines to Kripke structures, and obtained satisfaction of formulae in Moore machines by taking advantage of this map. We did not obtain results about assume reasoning model checking, however it seems that the categorial work presented here can be easily continued.

Bibliography

[AHS90] J. Adamek, H. Herrlich, and G. Strecker. *Abstract and concrete categories.* John Wiley and Sons, Inc., 1990.

[AS94] Uselton Andrew and Scott Smolka. A compositional semantics for statecharts using labeled transition systems. Technical report, State University of New York at Stony Brook, 1994.

[Boo94] Grady Booch. *Object-oriented analisys and design.* The Benjamin Cummings, Inc., 1994.

[BW90] M. Barr and C. Wells. *Category theory for computing science.* Prentice Hall, 1990.

[Cal95] C. Caleiro. Comunidades distribuidas de objectos. Master's thesis, Instituto Superior Tecnico, 1995.

[CES86] E. Clarke, E. Emerson, and A. Sistla. Concurrent systems using temporal logic specification. *ACM Transactions on Programming Languages and Systems*, 1986.

[GB84] J. Goguen and R. Burstall. Introducing intitutions. *Proceedings of the Logics of Programming Workshop*, 1984.

[GB92] J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of ACM*, 1992.

[Gol87] R. Goldblat. *Logics of time and Computation.* CSLI, 1987.

[Lon93] David E. Long. *Model checking, abstraction, and compositional verification.* PhD thesis, CMU, 1993.

[Mac71] S. MacLane. *Categories for the working mathematician.* Springer Verlag, 1971.

[Men96] P. Menezes. *Reificacao de objectos concorrentes.* PhD thesis, Instituto Superior Tecnico, 1996.

[Ram94] J. Ramos. Logica de certificao gnome. Master's thesis, Instituto Superior Tecnico, 1994.

[Rav95] A. Ravara. Semantica da linguagem gnome em tyco. Master's thesis, Instituto Superior Tecnico, 1995.

[RS95a] J. Ramos and A. Sernadas. A brief introduction to gnome. Technical report, Instituto Superior Tecnico, 1995.

[RS95b] J. Ramos and A. Sernadas. Gnome certification logic. Technical report, Instituto Superior Tecnico, 1995.

[SNR94] P. Scholz, D. Nazareth, and F. Regenburger. Mini-statecharts: A compositional way to model parallel systems. Technical report, Institut fur Informatik, 1994.

[SR94] A. Sernadas and J. Ramos. A linguagem gnome: Sintaxe, semantica e calculo. Technical report, Instituto Superior Tecnico, 1994.

[SSR96] A. Sernadas, C. Sernadas, and J. Ramos. A temporal logic approach to object certification. *Journal of Data and Knowledge Engineering*, 19:267–294, 1996.

[Tau82] Herbert Taub. *Circuitos digitais e microprocessadores.* McGraw-Hill, 1982.

[Wol95] P. Wolper. An introduction to model checking. Technical report, University of Liege, 1995.