Università di Udine
Dipartimento di Matematica e Informatica

Technical Report: 5-2010

CALCO Young Researchers Workshop
CALCO-jnr 2009

6 September 2009

Selected Papers

edited by

Magne Haveraaen, Marina Lenisa,
John Power, and Monika Seisenberger

Preface

CALCO brings together researchers and practitioners to exchange new results related to foundational aspects and both traditional and emerging uses of algebras and coalgebras in computer science. The study of algebra and coalgebra relates to the data, process and structural aspects of software systems.

This is a high-level, biennial conference formed by joining the forces and reputations of CMCS (the International Workshop on Coalgebraic Methods in Computer Science), and WADT (the Workshop on Algebraic Development Techniques). The first and second CALCO conferences were held 2005 in Swansea, Wales, and 2007 in Bergen, Norway. The third event took place 2009 in Udine, Italy.

The CALCO Young Researchers Workshop, CALCO-jnr, was a CALCO 2009 satellite event dedicated to presentations by PhD students and by those who completed their doctoral studies within the past few years. Attendance at the workshop was open to all – many CALCO conference participants attended CALCO-jnr and vice versa. The workshop had eleven contributions by authors from nine different countries and 50 participants.

CALCO-jnr presentations were, on the basis of submitted 2-page abstracts, selected by the programme committee. After the workshop, the authors of each presentation were invited to submit a full 10-15 page paper on the same topic. They were also asked to write anonymous reviews of papers submitted by other authors on related topics. Additional reviewing was organised and the final selection of papers was carried out by the programme committee. The volume of selected papers from the workshop is published as a technical report at Udine University. Authors will retain copyright, and are also encouraged to disseminate the results reported at CALCO-jnr by subsequent publication elsewhere.

The CALCO-jnr PC would like to thank the workshop participants, the reviewers, and the CALCO 2009 local organisers for their efforts and commitment which made this event very successful. The support of all sponsoring institutions is gratefully acknowledged.

March 2010

Magne Haveraaen
Marina Lenisa
John Power
Monika Seisenberger

Table of Contents

On the Decidability of Bigraphical Sorting

Giorgio Bacci Davide Grohmann

Dept. of Mathematics and Computer Science, University of Udine, Italy.
Via delle Scienze 206, I-33100 Udine, Italy.
{giorgio.bacci,grohmann}@dimi.uniud.it

Abstract. Bigraphical reactive systems are a general framework for ubiquitous and mobile computing, which is based on the concepts of location and channel connection. Despite their expressive power, in many cases "specialized version" of bigraphs have been defined by means of bigraphical sortings to fit precisely the computational model at hand. This paper investigates if (bigraphical) sortings are decidable, that is, decide if a bigraph belongs or not to some sorting. In general it is not the case, but we have found and proposed a decidable class of sortings by reducing the problem at issue to the matching problem for bigraphs.

1 Introduction

Bigraphical Reactive Systems (BRSs) [13] have been proposed as a promising meta-model for ubiquitous, mobile systems. The key feature of BRSs is that their states are *bigraphs*, semi-structured data which can represent at once both the (physical, logical) location and the connections of the components of a system. The dynamics of agents are represented by a set of rewrite rules.

Bigraphs and BRSs are very flexible. They have been used for representing many domain-specific calculi and models: programming languages, calculi for concurrency and mobility, context-aware systems and web-services [11,12,2,9,4].

Despite their expressive power, in many cases some "specialized versions" of bigraphs must be introduced. As an example, *binding bigraphs* [11] are necessary to encode the π-calculus: they allow for restricting name scope to only a portion of a bigraph's locations. Many other variants have been presented in literature, such as *homomorphic* and *many-one sortings* [13,12] or *kind bigraphs* [5].

Recently, Debois and coauthors [3] generalized the previous ad hoc constructions giving a definition of *sorted categories*. It turns out that such categories can be defined as particular functor $\mathcal{S} : \mathbf{X} \to \mathbf{C}$ which map the sorted category $\mathbf{X}$ into the original one $\mathbf{C}$ and are faithful and surjective on objects. Moreover, an interesting sub-class, named *predicate sortings*, of those functors is identified and defined by means of *decomposable predicates* P, such that $P(f \circ g)$ implies $P(f)$ and $P(g)$. Intuitively predicate sortings rule out all the undesired bigraphs, which are the ones that do not satisfy the predicate P.

One of the main advantages of using predicate sortings is that they provides a general construction of sortings which always sustain the behavioural theory of pure bigraphs, thus obviating the need to redevelop that theory for each

new application. Anyway, this technique has also a disadvantage: the systematic construction makes sorted categories very difficult to handle, due to the fact that their objects are defined as pairs of sets of morphisms from the original category, closed by prefix and suffix composition. Most of the difficulties arise when one wants to implement effectively such construction, indeed the sets of morphisms which form the objects of the category turn out to be undecidable when the chosen sorting predicate is undecidable.

In order to overcome these difficulties, but at the same time keeping the technique as general as possible (we do not want to define sortings by hand!), we propose to look at predicate sortings from a different point of view. The sorted category will not be constructed at all, but we will use sorting as a way of (automatically) checking if a morphism of the original category has a pre-image in the sorted category and, more importantly, if compositions in the non-sorted category are still possible in its sorted variant.

The aim of this paper is to determine a decidable but expressive subclass of predicate sortings, named *match predicate sortings*, for which there exists an effective algorithm to check if a bigraph belongs to the sorted category. To do so, we use the notion of *matching*, that is, all ill-formed bigraphs are ruled out if an unwanted pattern matches into some of their sub-components. Notably, those sortings capture a good variety of sortings in the literature.

Synopsis The paper is structured as follows. In Section 2 we recall the basics of bigraphs, sortings and bigraphical sortings. The undecidability of the (bigraphical) sorting problem is proved in Section 3, whilst in Section 4 a decidable class of sortings is proposed and analyzed. Many well-known sortings belong to our class as shown in Section 5. Finally, conclusions are in Section 6.

2 Preliminaries

2.1 Bigraphs

In this section we recall Milner's *bigraphs* [13]. Intuitively, a bigraph represents an open system, so it has an inner and an outer interface to "interact" with subsystems and the surrounding environment (see Fig. 1 for an example). The *width* of the outer interface describes the *roots*, that is, the various locations containing the system components; the width of the inner interface describes the *sites*, that is, the holes where other bigraphs can be inserted. On the other hand, the *names* in the interfaces describe the free links, that is end points where links from the outside world can be pasted, creating new links among nodes. We refer the reader to [13] for more details.

More formally, let $\mathcal{K}$ be a *signature* of controls, and $ar\colon \mathcal{K} \to \mathbb{N}$ be the arity function. The arity indexes the *ports* of a control.

Definition 2.1 (Interfaces). *An* interface *is a pair* $\langle m, X\rangle$ *where* m *is a finite ordinal (called* width*),* X *is a finite set of names.*

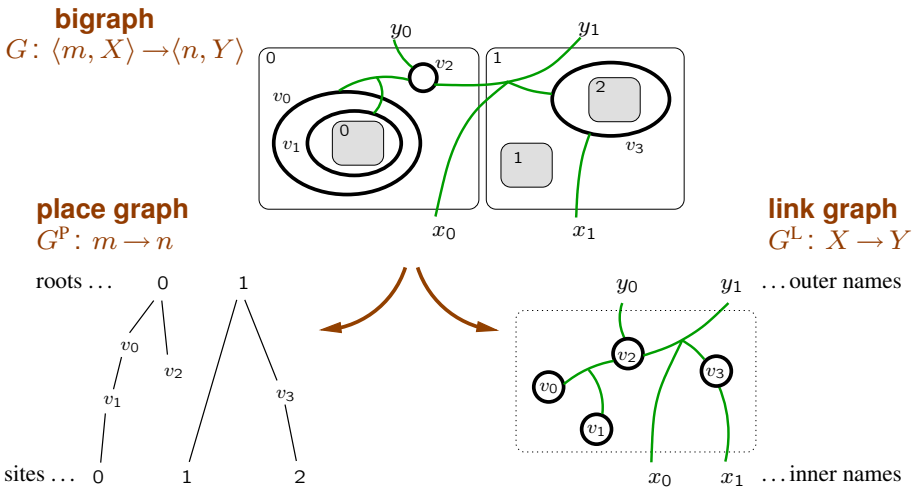


Fig. 1. A bigraph = a place graph + a link graph (picture taken from [13]).

In the following, we denote by $\uplus$ the union of already disjoint sets, that is, $S \uplus T \triangleq S \cup T$ if $S \cap T = \emptyset$, otherwise it is undefined.

Definition 2.2 (Bigraphs). *A* (pure) bigraph $G\colon \langle m, X\rangle \to \langle n, Y\rangle$ *is composed by a* place graph G^P *and a* link graph G^L *which describe the nesting of nodes and the (hyper-)links among nodes, respectively.*

$$G = (V, E, ctrl, G^P, G^L)\colon \langle m, X\rangle \to \langle n, Y\rangle \qquad \textit{(bigraph)}$$
$$G^P = (V, ctrl, prnt)\colon m \to n \qquad \textit{(place graph)}$$
$$G^L = (V, E, ctrl, link)\colon X \to Y \qquad \textit{(link graph)}$$

where V, E are the sets of nodes and edges respectively; $ctrl\colon V \to \mathcal{K}$ is the control map, *which assigns a control to each node; $prnt\colon m \uplus V \to V \uplus n$ is the (acyclic)* parent map *(often written also as $<$); $link\colon X \uplus P \to E \uplus Y$ is the* link map, *where $P = \sum_{v \in V} ar(ctrl(v))$ is the set of ports.*

Definition 2.3 (Bigraph category). *The category of bigraphs over a signature $\mathcal{K}$ (written* $\mathbf{Big}(\mathcal{K})$*) has interfaces as objects, and bigraphs as morphisms.*

Given two bigraphs $G\colon \langle m, X\rangle \to \langle n, Y\rangle$, $H\colon \langle n, Y\rangle \to \langle k, Z\rangle$, the composition $H \circ G\colon \langle m, X\rangle \to \langle k, Z\rangle$ is defined by composing their place and link graphs:

1. *the composition of $G^P\colon m \to n$ and $H^P\colon n \to k$ is defined as*

$$H^P \circ G^P = (V_G \uplus V_H, ctrl_G \uplus ctrl_H, (id_{V_G} \uplus prnt_H) \circ (prnt_G \uplus id_{V_H}))\colon n \to k;$$

2. *the composition of $G^L\colon X \to Y$ and $H^L\colon Y \to Z$ is defined as*

$$H^L \circ G^L = (V_G \uplus V_H, E_G \uplus E_H, ctrl_G \uplus ctrl_H, (id_{E_G} \uplus link_H) \circ (link_G \uplus id_{P_H}))\colon X \to Z.$$

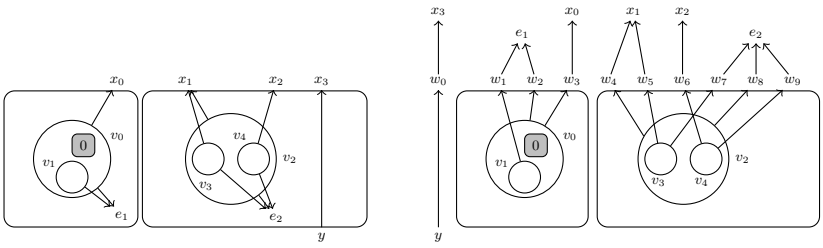


Fig. 2. A bigraph and its decomposition in discrete normal form

An important operation about (bi)graphs, is the *tensor product*. Intuitively, this operator puts "side by side" two bigraphs, i.e., given $G\colon \langle m, X\rangle \to \langle n, Y\rangle$ and $H\colon \langle m', X'\rangle \to \langle n', Y'\rangle$, their tensor product is $G \otimes H\colon \langle m+m', X \uplus X'\rangle \to \langle n+n', Y \uplus Y'\rangle$ defined when name sets X, X' and Y, Y' are pairwise disjoint.

As shown in [13], all bigraphs can be constructed by composition and tensor product from a set of *elementary bigraphs:*

- $1\colon \langle 0, \emptyset\rangle \to \langle 1, \emptyset\rangle$ is the barren (i.e., empty) root;
- $merge_n\colon \langle n, \emptyset\rangle \to \langle 1, \emptyset\rangle$ merges n roots into a single one;
- $\gamma_{m,n}\colon \langle m+n, \emptyset\rangle \to \langle n+m, \emptyset\rangle$ is a *symmetry*, that switches the first m roots with the following n roots.
- $/x\colon \langle 0, \{x\}\rangle \to \langle 0, \emptyset\rangle$ is a *closure*, that is it maps x to an edge;
- $y/X\colon \langle 0, X\rangle \to \langle 0, \{y\}\rangle$ substitutes the names in X with y, i.e., it maps the whole set X to y.
- Finally, $K_{\vec{x}}\colon \langle 1, \emptyset\rangle \to \langle 1, \{x_1, \ldots, x_n\}\rangle$ is a control which may contain other graphs, and it has ports linked to the name in $\vec{x} = x_1, \ldots, x_n$.

A bigraph is said a *renaming* if it is of the form $x_1/\{y_1\} \otimes \cdots \otimes x_n/\{y_n\}$ (abbreviated to $\vec{x}/\vec{y}$, where $\vec{x} = x_1, \ldots, x_n$ and $\vec{y} = y_1, \ldots, y_n$); a *permutation* if it is formed by composition and tensor product of symmetries; a *prime* when it has no inner names and its outer width is 1; finally a *discrete* when its link map is a bijection.

Bigraphs can be given always in *discrete normal form* [13]. In this case, the bigraph is split into two main components: *wirings* dealing with connections and *discrete bigraphs* describing only the nesting of nodes. The latters can be further decomposed using the algebra of bigraphs into the elementary parts: nodes, permutations and merging of roots. An example is shown in Fig. 2.

Proposition 2.4 (Discrete Normal Form, [13, Proposition 8.15]). *Every bigraph G can be expressed uniquely (up-to iso) as*

$$G = (id_n \otimes \omega) \circ D$$

where ω is a wiring and D is discrete. Further every discrete D may be factored uniquely (up-to iso) as

$$D = \alpha \otimes ((P_0 \otimes \cdots \otimes P_{n-1}) \circ \pi)$$

where α is a renaming, each P_i prime and discrete, and π is a permutation.

2.2 Sortings

As already stated, when one is adopting the bigraphical framework for defining algebraic models or programming languages, it turns out that such framework is too general and one has to discipline it with some constraints to fit precisely the problem at hand.

To this end, general and powerful techniques, named *sortings*, have been developed by Debois and coauthors in [3].

Definition 2.5 (Sortings). *A* sorting *of a category* $\mathbf{C}$ *is a functor* $F : \mathbf{X} \to \mathbf{C}$*, that is faithful and surjective on objects. We call* $\mathbf{X}$ sorted category*.*

Intuitively, a sorting functor F defines $\mathbf{X}$ by refining the category $\mathbf{C}$. The objects (i.e., interfaces) of $\mathbf{X}$ carry more information than the original ones, thus morphism (i.e., system) composition turns out to be finer-grained. This yields back a category $\mathbf{X}$ where morphisms are more informative than those in $\mathbf{C}$ in the sense that, some compositions in $\mathbf{C}$ no longer hold in $\mathbf{X}$.

Due to the very general nature of sorting refinements needed by each particular application, it could be tricky to construct a sorting directly by hand using the definition above. To avoid this, it is convenient to use predicates to characterize unwanted systems by means of predicates which describes their structure. When the set of ill-systems is defined, it is easy to characterize the good-ones: a systems is in the sorted category if and only if it does not "contain" an ill-one as subsystem. Formally,

Definition 2.6 (Decomposable predicate). *A* predicate P *on morphisms of a category* $\mathbf{C}$ *is* decomposable *if and only if it reflects identities and* $P(f \circ g)$ *implies* $P(f)$ *and* $P(g)$*.*

Notice that every decomposable predicate holds on identities.

Notably, the class of decomposable predicates can be characterize as those morphisms that disallow factorization by a given set of morphisms.

Theorem 2.7 (Factorization, [3, Proposition 14]). *Let* $\mathbf{C}$ *be a category. A predicate* P *on morphisms of a category* $\mathbf{C}$ *is* decomposable *if and only if there exists a set of* $\mathbf{C}$*-morphisms* Φ *such that* $P(f)$ *holds if and only if for any* g, ψ, h $f = g \circ \psi \circ h$ *implies* $\psi \notin \Phi$*.*

Interestingly, the Factorization Theorem 2.7 defines a connection with BiLog [6], a spatial logic for bigraphs. Indeed, given a BiLog formula ϕ which characterizes a set Φ of unwanted bigraphs, the formula $(\neg\phi)^{\forall\circ}$ defines all the bigraphs G such that $G = H \circ \psi \circ F$ implies $\psi \notin \Phi$. Obviously, all bigraphs satisfying the formula above are decomposable.

In [3] it is also given a method to systematically construct a well-behaved sorting for any decomposable predicate.

Definition 2.8 (Predicate Sorting). *Let* $\mathbf{C}$ *be a category, and let* P *be a decomposable predicate on the morphisms of* $\mathbf{C}$*. The* predicate sorting $\mathcal{S}_P : \mathbf{X} \to \mathbf{C}$ *is defined as follows. The category* $\mathbf{X}$ *has pairs* (X, Y) *as objects, where,*

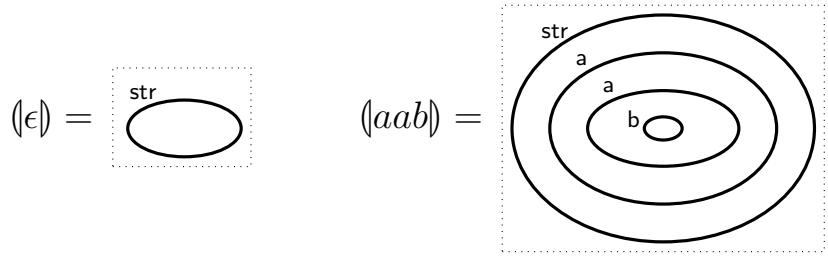


Fig. 3. Endcoding of strings in **Big**

for some object C of $\mathbf{C}$, X is a set of $\mathbf{C}$-morphisms with codomain C and Y is a set of $\mathbf{C}$-morphisms with domain C, subject to the following conditions.

$$\begin{array}{lll} id_C \in X & f \in X \cup Y \Rightarrow P(f) & g \circ f \in X \Rightarrow g \in X \\ id_C \in Y & f \in X,\ g \in Y \Rightarrow P(g \circ f) & g \circ f \in Y \Rightarrow f \in Y \end{array}$$

There is a morphism $f : (X, Y) \to (U, V)$ whenever the following holds.

$$f \in Y \cap U \qquad x \in X \Rightarrow f \circ x \in U \qquad v \in V \Rightarrow v \circ f \in Y$$

For such a construction, the following result holds:

Proposition 2.9. *Let P be a decomposable predicate on $\mathbf{C}$. The image of the predicate sorting $\mathcal{S}_P$ is precisely the set of morphisms satisfying P.*

Clearly, all the above definitions and results apply naturally to the bigraph category and they have been extensively used in literature to get rid of bigraphs that are meaningless for the application at hand. That is, most sortings exists solely to impose a predicate on the morphisms in the category of (pure) bigraphs.

3 Undecidability of bigraphical sortings

In this section we focus on some undecidability issues about predicate sortings, and in particular for the case of bigraphical sortings.

What we are really interested in is not the decidability of the construction of a sorted category, but of the problem of checking if a given morphism f from the base category has a pre-image in the sorted category. Fortunately, when a predicate sorting $\mathcal{S}_P \colon \mathbf{X} \to \mathbf{C}$ is used the existence of such pre-image $x = \mathcal{S}_P(f)$ in the sorted category $\mathbf{X}$ is garanteed whenever $P(f)$ holds, by means of Proposition 2.9. Decidability cannot be assumed nor for general predicate P neither for decomposable predicates over bigraphs.

Proposition 3.1. *Not all decomposable predicates over* **Big** *are decidable.*

Proof. The proof takes advantage of the characterization of a decomposable predicate given in Theorem 2.7. Let $\mathcal{K} = \{\mathsf{str}, \mathsf{a}, \mathsf{b}\}$ a bigraphical signature of

three 0-arity controls. Any word $w \in \{a, b\}^*$ in the two letters alphabet $\{a, b\}$ can be represented in $\mathbf{Big}(\mathcal{K})$ by means of the two encodings $(\!|\cdot|\!)\colon \{a, b\}^* \to \mathbf{Big}(\mathcal{K})$ and $[\![\cdot]\!]\colon \{a, b\}^* \to \mathbf{Big}(\mathcal{K})$ defined as follows (see also Fig. 3):

$$(\!|w|\!) = \mathsf{str} \circ [\![w]\!] \qquad [\![\epsilon]\!] = 1 \quad [\![a \cdot w]\!] = \mathsf{a} \circ [\![w]\!] \quad [\![b \cdot w]\!] = \mathsf{b} \circ [\![w]\!]\,.$$

Let $L \subseteq \{a, b\}^*$ be a language, and $\Phi_L = \{(\!|w|\!) \mid w \in L\}$ the set of morphisms that bigraphically represents L in $\mathbf{Big}(\mathcal{K})$. By the Factorization Theorem 2.7, the set Φ induces a decomposable predicate

$$P_L = \{f \text{ morphism of } \mathbf{Big}(\mathcal{K}) \mid \forall g, \phi, h.\ f = g \circ \phi \circ h \Rightarrow \phi \notin \Phi_L\}\,.$$

The problem of deciding $P_L(f)$ is reduced to check if f has no occurrences of $(\!|w|\!) \in \Phi_L$, which amounts to verify $w \in L$. If the given language L is not recursive, then the decomposable predicate is undecidable. □

Theorem 3.2. *Let $\mathcal{S}_P\colon \mathbf{X} \to \mathbf{C}$ be a predicate sorting over a decomposable predicate P. The problem of checking if a given morphism f in $\mathbf{C}$ has a pre-image $x = \mathcal{S}_P(f)$ in the sorted category $\mathbf{X}$ is undecidable.*

Proof. It follows immediately from Propositions 2.9 and 3.1. □

As corollary, it arises that also the construction of a predicate sorted category is not decidable, hence, more in general, the construction of a sorting.

Corollary 3.3. *The construction of a sorting, in general, is not decidable.*

Looking at the Definition 2.8, it is obvious that an exhaustive construction of the a predicate sorting category is unfeasible (one must quantifies on all morphisms to construct an object of the sorted category). Instead, given a decomposable predicate P, it results more convenient to look at the problem of checking if a given morphism has a pre-image in the sorted category. The key idea behind this approach is to *simulate* the existence of the sorted category, checking that each morphism that comes into play belongs to the sorted category. This can be done simply checking that the morphism satisfies the decomposable predicate P (note that this must be done only when a composition is performed).

Such approach is certainly feasible, but we want to do more: a general algorithm that works independently from the predicate P, in the sense that it has no need to be redeveloped every time if the predicate P changes. To this end, the next section introduces a decidable sub-class of predicate sortings, named *match predicate sortings*.

4 Match predicate sortings

In this section, we introduce a characterization of a decidable class of bigraphical sortings, which turns out to be a proper subclass of the predicate sortings.

The main idea behind our characterization is based on the fact that there is always a possible decomposition of a bigraph G in elementary bigraphs, and such

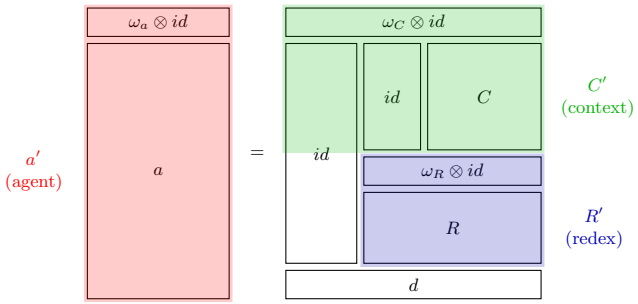


Fig. 4. A valid matching sentence.

decomposition is finite and computable. Indeed, every bigraph can be expressed uniquely in discrete normal form. Therefore checking if there is an occurrence of an ill-formed bigraph inside another one can be seen as trying to match the first into the latter. Such observation suggests that the problem at issue can be reduce to the matching of bigraphs: a bigraph G has a match into a bigraph H if and only if $H = F \circ (G \otimes id_X) \circ E$ for some name set X and bigraphs F, E.

Definition 4.1 (Matching Sentence). *A* matching sentence *is a 7-tuple relation among wirings and bigraphs, written* $\omega_a, \omega_R, \omega_C \vdash a, R \hookrightarrow C, d$*, where* $\omega_a, \omega_R, \omega_C$ *are wirings, and* a, R, C, d *are discrete bigraphs and* a, d *are ground.*

A matching sentence $\omega_a, \omega_R, \omega_C \vdash a, R \hookrightarrow C, d$ *is* valid *if and only if*

$$(id \otimes \omega_a) \circ a = (id \otimes \omega_C) \circ (C \otimes id) \circ (id \otimes (id \otimes \omega_R) \circ R) \circ d\,.$$

A schematic example of a valid matching sentence is shown in Fig. 4.

The matching of bigraph is decidable as shown in [1], where an effective algorithm has been proposed and studied. It is based on the Definition 4.1 of matching sentence and on a set of inference rules which operates on those sentences. Intuitively, the algorithm as first step transforms the agent and the redex in their respective discrete normal forms. Then it inductively decomposes and simplifies the structure of their components by picking a rule from the system. At each step, the parts of the agent which do not belong to the redex are "added" to the parts representing the context in the matching sentence. The algorithm terminates when a match is found, i.e., the redex is completely consumed by matching all its parts inside the agent. The remaining part of the agent are the parameters of the redex. Due to lack of space, we refer to [1] for more details.

Notice that the agent is forced to be ground (i.e., without sites, and inner names) in the previous algorithm. But it is not a limitation for our purpose, indeed we are interested only on the *existence* of a match. So, we can turn a non-ground bigraph into a ground one by filling the sites with empty (i.e., barren) roots and the inner names with idle names. More importantly, if a match exists into the ground bigraph then it must exist also into the non-ground one.

This scenario suggests the definition of a family of decomposable predicates based on the match concept.

Definition 4.2 (Match predicate). *Let $\mathcal{R}$ be a recursive set of redexes. We say that $P_{\mathcal{R}}$ is a* match predicate *with respect to the set $\mathcal{R}$, if for every (ground) bigraph g, $P_{\mathcal{R}}(g)$ holds if and only if every $R \in \mathcal{R}$ does not have a match into g, that is $\omega_g, \omega_R, \omega_C \vdash g, R \hookrightarrow C, d$ is not a valid matching sentence.*

We avoid to write the subscript $\mathcal{R}$, when can be understood from the context.

The following proposition holds straightforwardly.

Proposition 4.3. *Any match predicate is a decomposable predicate.*

Proof. Let $\mathcal{R}$ be a set of redexes and P its match predicate. Now, suppose by absurdity that P is not decomposable. So there exist two bigraphs such that $P(G \circ H)$ holds but one between $P(G)$ or $P(H)$ does not. Suppose $P(H)$ does not hold (the other case is analogous), this means that there exist C, d such that $H = (id \otimes C) \circ (id \otimes R) \circ d$ is a valid matching sentence for some $R \in \mathcal{R}$. But this means that $G \circ H = (G \circ (id \otimes C)) \circ (id \otimes R) \circ d$ is a valid matching sentence for $G \circ H$ with respect to R, but this is absurd by hypothesis. □

Hence, the class of match predicates is a *proper subclass* of decomposable predicates, and it is decidable by means of the matching algorithm.

Proposition 4.4 (Decidability). *Any match predicate is decidable.*

Proof. Direct consequence of the decidability result for the bigraphical matching problem [8].

Moreover, those results suggest a way to specialize the Factorization Theorem 2.7 for decomposable predicates in the following sense: given a recursive set of bigraphs M, the set Φ of unwanted bigraphs is defined from M as $\Phi = \{m \otimes id_X \mid m \in M \wedge X \text{ is a set of names}\}$.

Theorem 4.5 (Factorization). *A predicate P is a* match predicate *if and only if there exists a recursive set of morphisms M such that $P(f)$ holds if and only if for any g, ψ, h morphisms and any set of names X we have $f = g \circ (\psi \otimes id_X) \circ h$ implies $\psi \notin M$.*

Proof. Direct consequence of Proposition 4.3 and Theorem 2.7. □

In this way, deciding if a bigraph G is well-sorted is reduced to decide if no $m \in M$ has a match into G. Moreover, we can use the Proposition 4.3 in combination with the Definition 2.8 to compute the bigraphical sorted category. We call those class of sortings *match predicate sortings*.

There are some interesting observations to consider. To define a decidable sorting we must enforce that the set M is *recursive*, indeed such requirement is essential to be able to decide if an element belongs or not to some set. Whilst, to guarantee the meaningfulness of the sorting, the set M must contain no identities, otherwise (almost) all bigraphs are sorted out. Finally, our sortings work up-to tensor product with identities, i.e., the unwanted bigraphical structures are "homset independent", indeed a match can be found in any context, so we cannot force the decomposition to work only with some particular interfaces.

Another interesting fact, due to Theorem 4.5 and Proposition 4.3, is that there is again a connection with BiLog [6] as for predicate sortings. But in our case, the formulae must encode the fact that the redexes are matched up-to tensoring with an identity.

5 Sortings in literature and their decidability

In order to investigate the expressive power of our sortings, here we analyze some of the sortings already introduced in literature. Specifically, we focus our attention on some predicate sortings shown in [7, Table 6.1], to be replaceable by a predicate sorting. In particular, we consider *homomorphic sortings* [13] and the bigraph's variant known as *local bigraphs* [14]. Here we show that each decomposable predicate used in [7] can be characterized as a match predicate of Definition 4.2, then the construction of the sorted category remains unchanged because match predicates are decomposable by Proposition 4.3.

5.1 Homomorphic sortings

Firstly, we recall the definition of homomorphic sortings given in [13] with the variants of [7][1]. To do so, we start giving the definition of place-sorted bigraph.

Definition 5.1 (Place-sorted interface). *Let Θ be a set of sorts. An interface $I = \langle m, X\rangle$ is* Θ-place-sorted *if it is enriched by ascribing a sort to each place $i \in m$. If I is place-sorted, we denote its underlying unsorted interface by $U(I)$.*

We denote by $\mathbf{Big}(\mathcal{K}, \Theta)$ the (s-)category in which the objects are place-sorted interfaces, and each morphism $G : I \to J$ is a bigraph $G : U(I) \to U(J)$.

Such definition refines only the objects of the bigraph category, the next is cutting down some morphisms (i.e., bigraphs).

Definition 5.2 (Place-sorting). *A* place-sorting *is a triple $\Sigma = \{\mathcal{K}, \Theta, \Phi\}$, where Φ is a condition on the place graph of Θ-sorted bigraphs over $\mathcal{K}$. The condition Φ must be satisfied by the identity and preserved by composition and tensor product.*

A bigraph in $\mathbf{Big}(\mathcal{K}, \Theta)$ is Σ-place-sorted if it satisfies Φ. The Σ-sorted bigraphs from a sub-(s-)category of $\mathbf{Big}(\mathcal{K}, \Theta)$ denoted by $\mathbf{Big}(\Sigma)$.

Notably, Milner shows [13, Proposition 10.3] that U can be extended to a functor $U : \mathbf{Big}(\Sigma) \to \mathbf{Big}(\mathcal{K}, \Theta)$ which is faithful and surjective on objects, in other words it is a sorting by Definition 2.5.

Due to the very general nature of place-sorting, Milner defines a particular class of such sortings, named *homomorphic sortings*.

[1] The variants are quite technical and do not change the resulting sorted categories.

Definition 5.3. *A place-sorting $\Sigma = \{\mathcal{K}, \Theta, \Phi\}$ is an* homomorphic sorting *if the condition Φ assigns a sort $\theta \in \Theta$ to each control in $\mathcal{K}$ by means of a surjective function sort : $\mathcal{K} \to \Theta$ and it also defines a parent map $prnt_\Theta \colon \Theta \to \Theta$ over sorts. (We impose that Θ has a least two elements[2].)*

In a bigraph G, via its control map, the sort assignment to $\mathcal{K}$ determines a sort for each node. The Φ requires that, for each site or node w in G with sort θ:

1. *if $prnt_G(w)$ is a node then its sort is $prnt_\Theta(\theta)$;*
2. *if $prnt_G(w)$ is a root then its sort is θ.*

Notice that homomorphic sortings capture many sortings proposed in literature, such as the ones used in Jensen's PhD Thesis [10] to encode some π-calculi [15].

In order to construct a predicate from a homomorphic sorting, it is sufficient to restrict the condition of Φ to consider just 1. and dropping 2., indeed we can focus only on constraining the internal components (i.e., nodes) of the bigraph. The roots belong to the interfaces, and those are refined automatically by the predicate sortings (see Definition 2.8).

So, the predicate constructed by Debois in [7] is the following.

Definition 5.4. *Let $\Sigma = \{\mathcal{K}, \Theta, \Phi\}$ be a homomorphic sorting, and let $prnt_\Theta$ be the parent maps on sorts defined by Φ. The predicate P_Σ holds on a bigraph G if and only if whenever the control of a node v in G has sort $\theta \in \Theta$ and $w = prnt_G(v)$ is a node, then the control of w has sort $prnt_\Theta(\theta)$.*

On such definition it is not difficult to yield out a "negative" version of the predicate by means of the Factorization Theorem 2.7: the set of unwanted bigraphs Φ can be construct by complement the condition 1. This observation also suggests a simple way of constructing a match predicate by means of our Factorization Theorem 4.5.

Definition 5.5. *Let $\Sigma = \{\mathcal{K}, \Theta, \Phi\}$ be a homomorphic sorting, and let $prnt_\Theta$ be the parent maps on sorts defined by Φ. The match predicate $P(M_\Sigma)$ for Σ can be defined on the set of bigraphs M_Σ below and by using the Theorem 4.5.*

$$M_\Sigma \triangleq \{K_{\vec{x}} \circ H_{\vec{y}} \mid K, H \in \mathcal{K} \wedge prnt_\Theta(sort(H)) \neq sort(K)\} \tag{1}$$

It is easy to prove that the meaning of the two predicates coincides, indeed if a match exists condition 1. is violated, otherwise it does not hold.

It is important to notice for the decidability of the sorting, that the set M_Σ defined in the equation (1) is actually finite.

The following result follows directly from the above considerations.

Proposition 5.6. *Homomorphic sortings correspond to match predicate sortings over the predicate M_Σ.*

Proof. It follows immediately from the characterization given in [7, Section 6.3] and by the fact that $P_\Sigma = M_\Sigma$. □

[2] Otherwise the homomorphic sorting sorts out no bigraph, hence it is useless.

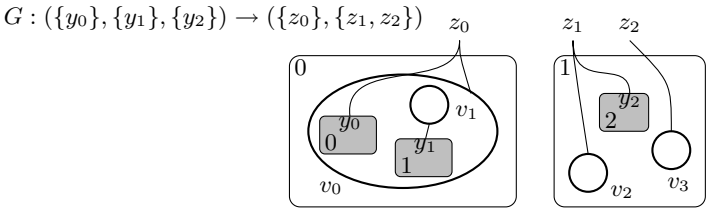


Fig. 5. An example of a local bigraph.

5.2 Local bigraphs

In this section firstly we recall Milner's *local bigraphs* [14] and then we discuss how to use a match predicate sorting on (pure) bigraphs to catch local bigraphs.

Intuitively, a local bigraph is like a standard bigraph but it has names which are deeply connected with placing, i.e., there is a precise scoping rule: every linking must respect the nesting of nodes. An example of a local bigraph is shown in Fig. 5.

Let $\mathcal{K}$ be a *binding signature* of controls, and $ar : \mathcal{K} \to \mathbb{N} \times \mathbb{N}$ be the arity function. The arity pair (h, k) (often written as $h \to k$) consists of the *binding arity* h and the *free arity* k, indexing respectively the binding ports and the free ports of a control.

Definition 5.7. *A* local interface *is a list* $(X_0, \ldots, X_{n-1})$*, where* n *is the* width *and* X_i*s are disjoint sets of names.* X_i *represents the names located at* i*.*

Definition 5.8. *A* local bigraph $G : (\vec{X}) \to (\vec{Y})$ *is defined as a (pure) bigraph* $G^u : \langle |\vec{X}|, \bigcup \vec{X} \rangle \to \langle |\vec{Y}|, \bigcup \vec{Y} \rangle$ *satisfying certain locality conditions.*

G^u *is defined much like as in Definition 2.2, the unique difference is in the link map: let* $P = \sum_{v \in V} \pi_1(ar(ctrl(v)))$ *be the set of ports and let* $B = \sum_{v \in V} \pi_2(ar(ctrl(v)))$ *be the set of bindings (associated to all nodes), the* link map *is link* $: X \uplus P \to E \uplus B \uplus Y$*.*

The locality conditions are the following:

1. *if a link is bound, then its inner names and ports must lie within the node that binds it;*
2. *if a link is free, with outer name* x*, then* x *must be located in every region that contains any inner name or port of the link.*

Definition 5.9. *The category* $\mathbf{Lbg}(\mathcal{K})$ *of local bigraphs over a signature* $\mathcal{K}$ *has local interfaces as objects, and local bigraphs as morphisms.*

Composition and tensor product are defined as for (pure) bigraphs.

In [7] it is given the predicate that follows, and it is proven that predicate sortings can be replaced with the category of local bigraphs.

Definition 5.10 ([7, Definition 6.22]). *Let* Σ *be a binding signature. Define* P_Σ *to be the predicate on the morphisms of* $\mathbf{Big}(U(\Sigma))$ *given by* $P_\Sigma(f)$ *if and only if in* f

"any port that is a peer of a binding port lies beneath that binding port".

It is straightforward to prove that such predicate is decomposable, but we want to characterize it as a matching predicate, that is giving a recursive set of unwanted redex patterns.

Fortunately the predicate P_Σ is very simple to falsify, indeed it is enough to find a match of a redex with the following form:

$$w/\{x_i, y_j\} \circ (K_{\vec{x}} \parallel N_{\vec{y}}) \tag{2}$$

where the i-th port of a control K is a binding port, for some control $K, N \in \Sigma$. Intuitively, if a bigraph has a match for a redex with the form in (2), that match is a counterexample for P_Σ (the binding port i of the K-node has as peer a port j of another node that is not beneath the K-node).

Now we can define the binding match predicate.

Definition 5.11. *Let Σ be a binding signature and let $\mathcal{R}_{bind}$ be the following set of bigraphs:*

$$\mathcal{R}_{bind} = \{w/\{x_i, y_j\} \circ (K_{\vec{x}} \parallel N_{\vec{y}}) \mid K, M \in \Sigma \text{ and } i \text{ is a binding port in } K\}$$

Define M_Σ as the match predicate defined on the (recursive) set of redexes $\mathcal{R}_{bind}$.

Proposition 5.12. *The category of local bigraphs corresponds to the one obtained by applying the match predicate sorting on M_Σ over the morphisms of* **Big**.

Proof. It follows immediately from the characterization given in [7, Section 6.4] and by the fact that $P_\Sigma = M_\Sigma$. $P_\Sigma \subseteq M_\Sigma$ can be proven noticing that any match of a redex from R_Σ in a bigraph f in $\mathbf{Big}(U(\Sigma))$ is a counterexample for $P_\Sigma(f)$; whereas $M_\Sigma \subseteq P_\Sigma$ follows immediately from the fact that R_Σ has a redex for any pair of controls in Σ, for any binding port. □

6 Conclusion

In this paper, we have investigated about the decidability problem of bigraphical sortings. In particular, we have shown the undecidability of Debois's predicate sortings, then we have identified a proper sub-class of them, named *match sortings*, which turns out to be decidable. For match sortings, we have proposed a characterization that induces the definition of an algorithm to check if a given morphism in the unsorted category has a pre-image into the sorted one, which holds independently from the chosen predicate. The algorithm is naturally based on the bigraphical matching problem, which has an effective solution procedure [1].

Notably, our match sortings preserve many interesting properties of predicate sortings, such as the possibility of describing unwanted bigraphs by means of BiLog formulae. Moreover, we have shown that the match sortings are powerful

enough to capture some important bigraphical sortings proposed in literature: *homomorphic sorting* and the well-known bigraph's variant of *local bigraph*.

As possible future work, we plan to investigate if other decidable classes of sortings exist and if there are other (possibly) more efficient algorithms to decide if a bigraph belongs to a sortings (remark that the matching problem for bigraphs is NP-complete). Finally, another interesting developing could be the analysis of the problem in a more general setting, not focusing only on bigraphs.

References

1. L. Birkedal, T. C. Damgaard, A. J. Glenstrup, and R. Milner. Matching of bigraphs. *Electr. Notes Theor. Comput. Sci.*, 175(4):3–19, 2007.
2. L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt, and H. Niss. Bigraphical models of context-aware systems. In L. Aceto and A. Ingólfsdóttir, editors, *Proc. FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2006.
3. L. Birkedal, S. Debois, and T. T. Hildebrandt. Sortings for reactive systems. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2006.
4. M. Bundgaard, A. J. Glenstrup, T. T. Hildebrandt, E. Højsgaard, and H. Niss. Formalizing higher-order mobile embedded business processes with binding bigraphs. In D. Lea and G. Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2008.
5. S. Ó. Conchúir. Kind bigraphs. *Electr. Notes Theor. Comput. Sci.*, 225:361–377, 2009.
6. G. Conforti, D. Macedonio, and V. Sassone. Spatial logics for bigraphs. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *Proc. ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 766–778. Springer, 2005.
7. S. Debois. *Sortings and Bigraphs*. PhD thesis, IT University of Copenhagen, 2008. `http://www.itu.dk/people/debois/pubs/thesis.pdf`.
8. A. Glenstrup, T. Damgaard, L. Birkedal, and E. Højsgaard. An implementation of bigraph matching. *IT University of Copenhagen*, 2007. `http://www.itu.dk/~tcd/docs/implBigraphMatching.pdf`.
9. D. Grohmann and M. Miculan. Reactive systems over directed bigraphs. In L. Caires and V. T. Vasconcelos, editors, *Proc. CONCUR 2007*, volume 4703 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 2007.
10. O. H. Jensen. *Mobile Processes in Bigraphs*. PhD thesis, University of Aalborg, 2008. To appear.
11. O. H. Jensen and R. Milner. Bigraphs and transitions. In *Proc. POPL*, pages 38–49, 2003.
12. J. J. Leifer and R. Milner. Transition systems, link graphs and petri nets. *Mathematical Structures in Computer Science*, 16(6):989–1047, 2006.
13. R. Milner. Pure bigraphs: Structure and dynamics. *Information and Computation*, 204(1):60–122, 2006.
14. R. Milner. Local bigraphs and confluence: Two conjectures. In *Proc. EXPRESS 2006*, volume 175(3) of *Electronic Notes in Theoretical Computer Science*, pages 65–73. Elsevier, 2007.
15. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

Modalities and Quantifiers in Institution Theory

Fabrice Barbier

Boole Centre for Research in Informatics,
University College Cork, Ireland
fgbarbier@gmail.com

Abstract. Introducing the concept of elements-based semantics, we investigate the concept of satisfaction within Institution Theory. Based on the elements used to define the satisfaction relation the concept of elements-based semantics allows us to model in a uniform way the two concepts of quantifier and modality.

Key words: Modality, Quantifier, Satisfaction Relation, Institution

When one has a closer look on how the satisfaction relation between formulae and models is defined in temporal/modal logics and predicate logics, it clearly appears that, in both cases, two steps are necessary in order to define the satisfaction relation. Propositional logic is the only logic whose satisfaction relation can be defined directly. But, for every other logic the validity of a given formula cannot be checked directly once given the semantical structure. One must first consider the elements of the semantical structure, *i.e.* the states/worlds or the interpretations of variables, and define the satisfaction of a formula with respect to these elements. Only then one can define the overall satisfaction of a given formula in the semantical structure.

In this paper we introduce the notion of *Elements-based Semantics* that will allow us to model these additional elements and investigate further the notion of satisfaction within Institution Theory. We will first show how it is possible to simulate the concept of interpretation of variables by simply playing tricks with the syntax while strictly staying in the limits of Institution Theory. This gives rise to the concept of *Internal Elements-based Semantics* ($i\mathfrak{EBS}$) and will allow us to build, from a given institution, the associated institution of open formulae.

But the concept of internal elements-based semantics is limited to semantics based on interpretation of variables. Its scope does not include modal logics. Taking inspiration on the concept of internal elements-based semantics, and in order to capture temporal/modal logics, we then generalize it to the concept of *(External) Elements-based Semantics* ($\mathfrak{EBS}$), introducing the framework of $\mathfrak{EBS}$-*Institution*. This framework simply refines the institutional framework by considering a certain set of states associated with every model. This makes the satisfaction relation more explicit than in Institution Theory, thus allowing a deeper investigation of its definition while remaining at a very abstract level.

In order to model quantifiers and modalities we finally extend the $\mathfrak{EBS}$-Institutions with a notion of abstract syntax, introducing the $\kappa\mathfrak{EBS}$-*Institutions*.

The association of both concepts of abstract syntax and elements-based semantics emphasize the similarities between the concepts of quantifiers and modalities. We believe this will allow a thorough study of both concepts and opens the door to a new approach to logic combination.

In the sequel proofs have been omitted for lack of space.

1 Institution

Institution Theory (*cf.* [GB92]) was introduced at the beginning of the 80's by J. Goguen and R. Burstall following their work on the semantics of the language Clear (*cf.* [BG80]). Aiming at doing "as much computing science as possible" independently of what the underlying logic may be, they achieve to characterize in a generic way the indispensable concepts for a theory to be an adequate framework for the conception of information systems.

1.1 Definition

An institution makes a distinction between signatures and sentences inside theories. Without giving any details about the structure of models or the structure of sentences, an institution focuses on the relationship between syntax and semantics, *i.e.* between sets of sentences and categories of models, in an abstract way. In a certain way an institution is an abstract logic laking all the inference mechanisms but that allows to formally capture the imprecise concept of a logical system from a model-theoretic point of view.

Definition 1 (Institution). *An* institution *is a 4-tuple* $(\mathcal{Sig}, \mathcal{Sen}, \mathcal{Mod}, \models)$ *such that:*

- $\mathcal{Sig}$ *is a* category*, objects of which are called* signatures*;*
- $\mathcal{Sen} : \mathcal{Sig} \rightarrow \mathcal{Set}$ *is a* functor *that associates every signature* $\Sigma \in |\mathcal{Sig}|$ *with the set* $\mathcal{Sen}(\Sigma) \in |\mathcal{Set}|$ *of all sentences built over* Σ*. Elements in* $\mathcal{Sen}(\Sigma)$ *are called* Σ-sentences*;*
- $\mathcal{Mod} : \mathcal{Sig} \rightarrow \mathcal{Cat}^{op}$ *is a* functor *that associates every signature* $\Sigma \in |\mathcal{Sig}|$ *with the category* $\mathcal{Mod}(\Sigma) \in |\mathcal{Cat}|$ *of models over* Σ*. Objects in* $\mathcal{Mod}(\Sigma)$ *are called* Σ-models*;*
- $\models$ *is a* family of binary relations $(\models_\Sigma)_{\Sigma \in |\mathcal{Sig}|}$ *such that for every signature* $\Sigma \in |\mathcal{Sig}|$, $\models_\Sigma \subseteq |\mathcal{Mod}(\Sigma)| \times \mathcal{Sen}(\Sigma)$ *is the* satisfaction relation of Σ-sentences in Σ-models*. We shall write* $\mathcal{M} \models_\Sigma \varphi$ *to indicate that the* Σ*-sentence* $\varphi \in \mathcal{Sen}(\Sigma)$ *holds in the* Σ*-model* $\mathcal{M}$*.*

Moreover, for every signature morphism $\sigma : \Sigma \rightarrow \Sigma'$*, every* Σ'*-model* $\mathcal{M}' \in |\mathcal{Mod}(\Sigma')|$ *and every* Σ*-sentence* $\varphi \in \mathcal{Sen}(\Sigma)$*, we have:*

$$\mathcal{M}' \models_{\Sigma'} \mathcal{Sen}(\sigma)(\varphi) \Leftrightarrow \mathcal{Mod}(\sigma)(\mathcal{M}') \models_\Sigma \varphi \tag{1}$$

The above formula is called *satisfaction condition.*

Notation 11 *We will note* $\mathcal{M} \not\models_\Sigma \varphi$ *if* $\mathcal{M} \models_\Sigma \varphi$ *is false.*

1.2 Examples

Logic deals with two main features: quantification and modality. This gives rise to four different cases: (i) a unique possible world and no quantification (PL); (ii) multiple possible worlds but no quantification (MPL); (iii) a unique possible world with quantification (FOL); (iv) multiple possible worlds and quantification (MFOL).

Example 1 (Propositional Logic (PL)). The institution of Propositional Logic is defined as follows:

- $\mathcal{S}ig_{\text{PL}}$ is the category $\mathcal{S}et$ of sets and functions between them (called morphisms). We call *propositional signatures* any object in $\mathcal{S}ig_{\text{PL}}$. An element of a signature $\Sigma \in |\mathcal{S}ig_{\text{PL}}|$ is called a *propositional variable*;
- $\mathcal{S}en_{\text{PL}} : \mathcal{S}ig_{\text{PL}} \to \mathcal{S}et$ is the functor that associates every signature $\Sigma \in |\mathcal{S}ig_{\text{PL}}|$ with the set $\mathcal{S}en_{\text{PL}}(\Sigma) \in |\mathcal{S}et|$ of *propositional Σ-sentences*, *i.e.* $\mathcal{S}en_{\text{PL}}(\Sigma)$ contains Σ and is closed under negation ($\neg$) and disjunction ($\vee$). It lifts signature morphisms to the level of sets of sentences;
- $\mathcal{M}od_{\text{PL}} : \mathcal{S}ig_{\text{PL}} \to \mathcal{C}at^{op}$ is the functor that associates every signature $\Sigma \in |\mathcal{S}ig_{\text{PL}}|$ with the category $\mathcal{M}od_{\text{PL}}(\Sigma) \in |\mathcal{C}at|$ of *Σ-valuations* and functions between them. Given a signature morphism $\sigma \in \mathcal{S}ig_{\text{PL}}(\Sigma, \Sigma')$, the σ-reduct $\mathcal{M}od_{\text{PL}}(\sigma)(v') \in |\mathcal{M}od_{\text{PL}}(\Sigma)|$ of a Σ'-valuation $v' \in |\mathcal{M}od_{\text{PL}}(\Sigma')|$ is the Σ-valuation $v = v' \circ \sigma$;
- $\models_{\text{PL}} = (\models^{\text{PL}}_{\Sigma})_{\Sigma \in |\mathcal{S}ig_{\text{PL}}|}$ is the family of binary relations such that for every signature $\Sigma \in |\mathcal{S}ig_{\text{PL}}|$, the relation $\models^{\text{PL}}_{\Sigma} \subseteq |\mathcal{M}od_{\text{PL}}(\Sigma)| \times \mathcal{S}en_{\text{PL}}(\Sigma)$ is defined in the following way (where $v \in |\mathcal{M}od_{\text{PL}}(\Sigma)|$):
  - $\forall p \in \Sigma$, $v \models^{\text{PL}}_{\Sigma} p$ iff $v(p) = 1$,
  - $\forall \varphi \in \mathcal{S}en_{\text{PL}}(\Sigma)$, $v \models^{\text{PL}}_{\Sigma} \neg\varphi$ iff $v \not\models^{\text{PL}}_{\Sigma} \varphi$,
  - $\forall \varphi, \psi \in \mathcal{S}en_{\text{PL}}(\Sigma)$, $v \models^{\text{PL}}_{\Sigma} \varphi \vee \psi$ iff $v \models^{\text{PL}}_{\Sigma} \varphi$ ou $v \models^{\text{PL}}_{\Sigma} \psi$.

Example 2 (Modal Propositional Logic (MPL)). The institution of Modal Propositional Logic is defined as follows:[1]

- $\mathcal{S}ig_{\text{MPL}}$ is the category $\mathcal{S}ig_{\text{PL}}$ of propositional signatures;
- $\mathcal{S}en_{\text{MPL}} : \mathcal{S}ig_{\text{MPL}} \to \mathcal{S}et$ is the functor that associates every signature $\Sigma \in |\mathcal{S}ig_{\text{MPL}}|$ with the set $\mathcal{S}en_{\text{MPL}}(\Sigma) \in |\mathcal{S}et|$ of *modal propositional Σ-sentences*, *i.e.* $\mathcal{S}en_{\text{MPL}}(\Sigma)$ contains the signature Σ and is closed under negation, disjunction and the modality $\Box$. It lifts signature morphisms to the level of sets of sentences;
- $\mathcal{M}od_{\text{MPL}} : \mathcal{S}ig_{\text{MPL}} \to \mathcal{C}at^{op}$ is the functor that associates every signature $\Sigma \in |\mathcal{S}ig_{\text{MPL}}|$ with the category $\mathcal{M}od_{\text{MPL}}(\Sigma) \in |\mathcal{C}at|$ of *Kripke Σ-models*, *i.e.* the category whose objects are pairs $(\mathfrak{E}, \mathfrak{R})$ where $\mathfrak{E}$ is a non-empty set of states and $\mathfrak{R} \subseteq \mathfrak{E} \times \mathfrak{E}$ is a binary relation over $\mathfrak{E}$ called *accessibility relation*. For

[1] We do not focus our attention at a specific modal logic. What we give here is a very general presentation of modal and temporal logics. These logics may differ in many different ways except they all consider a modality (or a temporal operator) as a connective.

every model $(\mathfrak{E}, \mathfrak{R})$ a mapping $\mathfrak{v} : \mathfrak{E} \times \Sigma \to \{0, 1\}$ is defined and associates every pair `(state,propositional variable)` with a truth value. Given a signature morphism $\sigma \in \mathcal{S}ig_{\text{MPL}}(\Sigma, \Sigma')$, the σ-reduct $\mathcal{M}od_{\text{MPL}}(\sigma)((\mathfrak{E}', \mathfrak{R}') \in |\mathcal{M}od_{\text{MPL}}(\Sigma)|$ of a Σ'-model $(\mathfrak{E}', \mathfrak{R}') \in |\mathcal{M}od_{\text{MPL}}(\Sigma')|$ is the Kripke Σ-model $(\mathfrak{E}, \mathfrak{R})$ such that $\mathfrak{E} = \mathfrak{E}'$ and:

- The accessibility relation is preserved, *i.e.* : $\forall(\eta'_1, \eta'_2) \in \mathfrak{E}' \times \mathfrak{E}'$,

$$\eta'_1 \mathfrak{R}' \eta'_2 \Rightarrow \mathcal{M}od_{\text{MPL}}(\sigma)(\eta'_1) \mathfrak{R} \mathcal{M}od_{\text{MPL}}(\sigma)(\eta'_2)$$

- For every propositional variable $p \in \Sigma$ and every state $\eta \in \mathfrak{E}$, we have:

$$\mathfrak{v}(p, \eta) = \mathfrak{v}'(\sigma(p), \eta)$$

– $(\models^{\text{MPL}}_{\Sigma})_{\Sigma \in |\mathcal{S}ig_{\text{MPL}}|}$ is the family of binary relations such that for every signature $\Sigma \in |\mathcal{S}ig_{\text{MPL}}|$, every Σ-sentence $\varphi \in \mathcal{S}en_{\text{MPL}}(\Sigma)$ and every Σ-model $(\mathfrak{E}, \mathfrak{R}) \in |\mathcal{M}od_{\text{MPL}}(\Sigma)|$, $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MPL}}_{\Sigma} \varphi$ if and only if for every $\eta \in \mathfrak{E}$ we have $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MPL}}_{\Sigma,\eta} \varphi$, where the relation $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MPL}}_{\Sigma,\eta} \subseteq \mathcal{S}en_{\text{MPL}}(\Sigma)$ is defined in the following way for every state $\eta \in \mathfrak{E}$:
  - $\forall p \in \Sigma$, $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MPL}}_{\Sigma,\eta} p$ iff $\mathfrak{v}(\eta, p) = 1$,
  - $\forall \varphi \in \mathcal{S}en_{\text{MPL}}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MPL}}_{\Sigma,\eta} \neg\varphi$ iff $(\mathfrak{E}, \mathfrak{R}) \not\models^{\text{MPL}}_{\Sigma,\eta} \varphi$,
  - $\forall \varphi \in \mathcal{S}en_{\text{MPL}}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MPL}}_{\Sigma,\eta} \Box\varphi$ iff for every $\eta' \in \mathfrak{E}$ such that $\eta \mathfrak{R} \eta'$, we have $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MPL}}_{\Sigma,\eta'} \varphi$,
  - $\forall \varphi, \psi \in \mathcal{S}en_{\text{MPL}}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MPL}}_{\Sigma,\eta} \varphi \vee \psi$ iff $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MPL}}_{\Sigma,\eta} \varphi$ or $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MPL}}_{\Sigma,\eta} \psi$.

Example 3 (First Order Logic). The institution of First Order Logic is defined as follows:

– $\mathcal{S}ig_{\text{FOL}}$ is the category of *unsorted first order signatures*, *i.e.* of pairs $(\mathcal{F}, \mathcal{R})$ where $\mathcal{F} = (\mathcal{F}^n)_{n \in \mathbb{N}}$ is a family of sets of function signs and $\mathcal{R} = (\mathcal{R}^n)_{n \in \mathbb{N}}$ is a family of sets of relation signs.[2] First order signature morphisms preserve both the structure of the pair $(\mathcal{F}, \mathcal{R})$ and the arities;

– $\mathcal{S}en_{\text{FOL}} : \mathcal{S}ig_{\text{FOL}} \to \mathcal{S}et$ is the functor that associates every signature $\Sigma \in |\mathcal{S}ig_{\text{FOL}}|$ with the set $\mathcal{S}en_{\text{FOL}}(\Sigma) \in |\mathcal{S}et|$ of the *unsorted first order Σ-sentences*,*i.e.* $\mathcal{S}en_{\text{FOL}}(\Sigma)$ contains the set of atomic Σ-sentences $\mathcal{A}tom_{\Sigma}$[3] and is closed under negation, disjunction and the universal quantifier ($\forall$). It lifts signature morphisms to the level of sets of sentences;

– $\mathcal{M}od_{\text{FOL}} : \mathcal{S}ig_{\text{FOL}} \to \mathcal{C}at^{op}$ is the functor that associates every signature $(\mathcal{F}, \mathcal{R}) \in |\mathcal{S}ig_{\text{FOL}}|$ with the category $\mathcal{M}od_{\text{FOL}}((\mathcal{F}, \mathcal{R})) \in |\mathcal{C}at|$ of *first order $(\mathcal{F}, \mathcal{R})$-structures* $\mathcal{M}$, *i.e.* carrier sets $M \in |\mathcal{S}et|$ equipped with a function $f^{\mathcal{M}} : M^n \to M$ for every function sign $f \in \mathcal{F}^n$ and a relation $r^{\mathcal{M}} \subseteq M^n$ for every relation sign $r \in \mathcal{R}^n$. Given a signature morphism $\sigma \in \mathcal{S}ig_{\text{FOL}}(\Sigma, \Sigma')$, the σ-reduct $\mathcal{M}od_{\text{FOL}}(\sigma)(\mathcal{M}') \in |\mathcal{M}od_{nomFOL}(\Sigma)|$ of a Σ'-structure $\mathcal{M}' \in |\mathcal{M}od_{\text{FOL}}(\Sigma')|$ is defined as follows:
  - $\mathcal{M}od_{\text{FOL}}(\sigma)(M') = M'$,

[2] Both function signs and relation signs are equipped with an arity $n \in \mathbb{N}$.

[3] Atomic Σ-sentences are of the form $r(t_1, \ldots, t_n)$ where $r \in \mathcal{R}^n$ is a relation in $\mathcal{R}^n$ and $t_1, \ldots, t_n \in T_{\Sigma}(\mathcal{X})$ are Σ-terms ($\mathcal{X}$ is a set of unsorted first order variables).

  - $f^{\mathcal{M}od_{\mathrm{FOL}}(\sigma)(\mathcal{M}')} = \sigma_{\mathcal{F}}(f)^{\mathcal{M}'}$ for every function sign $f \in \mathcal{F}$,
  - $r^{\mathcal{M}od_{\mathrm{FOL}}(\sigma)(\mathcal{M}')} = \sigma_{\mathcal{R}}(r)^{\mathcal{M}'}$ for every relation sign $r \in \mathcal{R}$.
- $(\models^{\mathrm{FOL}}_{\Sigma})_{\Sigma \in |\mathcal{S}ig_{\mathrm{FOL}}|}$ is the family of binary relations such that for every signature $\Sigma \in |\mathcal{S}ig_{\mathrm{FOL}}|$, every Σ-sentence $\varphi \in \mathcal{S}en_{\mathrm{FOL}}(\Sigma)$ and every Σ-model $\mathcal{M} \in |\mathcal{M}od_{\mathrm{FOL}}(\Sigma)|$, $\mathcal{M} \models^{\mathrm{FOL}}_{\Sigma} \varphi$ if and only if for every interpretation $\nu \in M^{\mathcal{X}}$ we have $\mathcal{M} \models^{\mathrm{FOL}}_{\Sigma,\nu} \varphi$, where the relation $\mathcal{M} \models^{\mathrm{FOL}}_{\Sigma,\nu} \subseteq \mathcal{S}en_{\mathrm{FOL}}(\Sigma)$ is defined in the following way for every interpretation $\nu \in M^{\mathcal{X}}$:
  - $\forall r \in \mathcal{R}^n, \forall t_1, \ldots, t_n \in T_{\Sigma}(\mathcal{X})$, $\mathcal{M} \models^{\mathrm{FOL}}_{\Sigma,\nu} r(t_1, \ldots, t_n)$ iff

$$(\nu(t_1), \ldots, \nu(t_n)) \in r^{\mathcal{M}}$$

  - $\forall \varphi \in \mathcal{S}en_{\mathrm{FOL}}(\Sigma)$, $\mathcal{M} \models^{\mathrm{FOL}}_{\Sigma,\nu} \neg\varphi$ iff $\mathcal{M} \not\models^{\mathrm{FOL}}_{\Sigma,\nu} \varphi$,
  - $\forall x \in \mathcal{X}, \forall \varphi \in \mathcal{S}en_{\mathrm{FOL}}(\Sigma)$, $\mathcal{M} \models^{\mathrm{FOL}}_{\Sigma,\nu} (\forall x)\varphi$ iff $\mathcal{M} \models^{\mathrm{FOL}}_{\Sigma,\nu'} \varphi$ for every interpretation ν' such that $\nu'(y) = \nu(y)$ for every variable $y \in \mathcal{X}$, except eventually x,
  - $\forall \varphi, \psi \in \mathcal{S}en_{\mathrm{FOL}}(\Sigma)$, $\mathcal{M} \models^{\mathrm{FOL}}_{\Sigma,\nu} \varphi \vee \psi$ iff $\mathcal{M} \models^{\mathrm{FOL}}_{\Sigma,\nu} \varphi$ or $\mathcal{M} \models^{\mathrm{FOL}}_{\Sigma,\nu} \psi$

Example 4 (Modal First Order Logic (MFOL)). The institution of Modal First Order Logic is defined as follows:

- $\mathcal{S}ig_{\mathrm{MFOL}} = \mathcal{S}ig_{\mathrm{FOL}}$ is the category of first order signatures;
- $\mathcal{S}en_{\mathrm{MFOL}} : \mathcal{S}ig_{\mathrm{MFOL}} \to \mathcal{S}et$ is the functor that associates every signature $\Sigma \in |\mathcal{S}ig_{\mathrm{MFOL}}|$ with the set $\mathcal{S}en_{\mathrm{MFOL}}(\Sigma) \in |\mathcal{S}et|$ of *modal first order Σ-sentences*, *i.e.* $\mathcal{S}en_{\mathrm{MFOL}}$ contains $\mathcal{A}tom_{\Sigma}$ is closed under negation, disjonction, universal quantification and the modality $\Box$. It lifts signature morphisms to the level of sets of sentences;
- $\mathcal{M}od_{\mathrm{MFOL}} : \mathcal{S}ig^{op}_{\mathrm{MFOL}} \to \mathcal{C}at$ is the functor that associates every signature $(\mathcal{F}, \mathcal{R}) \in |\mathcal{S}ig_{\mathrm{MFOL}}|$ with the category of *Kripke $(\mathcal{F}, \mathcal{R})$-models* $(\mathfrak{E}, \mathfrak{R})$ where $\mathfrak{E}$ is a set of states, each one of them being a first order $(\mathcal{F}, \mathcal{R})$-structure, and $\mathcal{R} \subseteq \mathfrak{E} \times \mathfrak{E}$ is a binary relation over the states called accessibility relation. Given a signature morphism $\sigma \in \mathcal{S}ig_{\mathrm{MFOL}}(\Sigma, \Sigma')$ and a Krikpe $(\mathcal{F}', \mathcal{R}')$-model $(\mathfrak{E}', \mathfrak{R}')$, the σ-reduct of $(\mathfrak{E}', \mathfrak{R}')$ is the Kripke $(\mathcal{F}, \mathcal{R})$-model $\mathcal{M}od_{\mathrm{MFOL}}(\sigma)((\mathfrak{E}', \mathfrak{R}'))$ such that $\mathcal{M}od_{\mathrm{MFOL}}(\sigma)(\mathfrak{E}') = \mathfrak{E}'$ and:
  - The accessibility relation is preserved,
  - For every state $\mathcal{M}' \in \mathcal{M}od_{\mathrm{MFOL}}(\sigma)(\mathfrak{E}')$, $\mathcal{M}od_{\mathrm{MFOL}}(\sigma)(\mathcal{M}')$ is the first order $(\mathcal{F}, \mathcal{R})$-structure $\mathcal{M}od_{\mathrm{MFOL}}(\sigma)(\mathcal{M}')$ such that:
    - $\mathcal{M}od_{\mathrm{MFOL}}(\sigma)(M') = M'$,
    - $f^{\mathcal{M}od_{\mathrm{MFOL}}(\sigma)(\mathcal{M}')} = \sigma_{\mathcal{F}}(f)^{\mathcal{M}'}$ for every function sign $f \in \mathcal{F}$,
    - $r^{\mathcal{M}od_{\mathrm{MFOL}}(\sigma)(\mathcal{M}')} = \sigma_{\mathcal{R}}(r)^{\mathcal{M}'}$ for every relation sign $r \in \mathcal{R}$.
- $\models_{\mathrm{MFOL}} = (\models^{\mathrm{MFOL}}_{\Sigma})_{\Sigma \in |\mathcal{S}ig_{\mathrm{MFOL}}|}$ is the family of binary relations such that for every signature $\Sigma \in |\mathcal{S}ig_{\mathrm{MFOL}}|$, every Σ-sentence $\varphi \in \mathcal{S}en_{\mathrm{MFOL}}(\Sigma)$ and every Σ-model $(\mathfrak{E}, \mathfrak{R}) \in |\mathcal{M}od_{\mathrm{MFOL}}(\Sigma)|$, $(\mathfrak{E}, \mathfrak{R}) \models^{\mathrm{MFOL}}_{\Sigma} \varphi$ if and only if for every state $\mathcal{M} \in \mathfrak{E}$ we have $(\mathfrak{E}, \mathfrak{R}) \models^{\mathrm{MFOL}}_{\Sigma,\mathcal{M}} \varphi$ where $(\mathfrak{E}, \mathfrak{R}) \models^{\mathrm{MFOL}}_{\Sigma,\mathcal{M}} \subseteq \mathcal{S}en_{\mathrm{MFOL}}(\Sigma)$ is defined, for evey state $\mathcal{M} \in \mathfrak{E}$ as $(\mathfrak{E}, \mathfrak{R}) \models^{\mathrm{MFOL}}_{\Sigma,\mathcal{M}} \varphi$ if and only if for every interpretation $\nu \in M^{\mathcal{X}}$ we have $\mathcal{M} \models^{\mathrm{MFOL}}_{\Sigma,\mathcal{M},\nu} \varphi$, where the relation $\mathcal{M} \models^{\mathrm{MFOL}}_{\Sigma,\nu} \subseteq \mathcal{S}en_{\mathrm{MFOL}}(\Sigma)$ is defined in the following way for every interpretation $\nu \in M^{\mathcal{X}}$:

- $\forall r \in \mathcal{R}, \forall t_1, \ldots, t_n \in T_\Sigma(\mathcal{X})$, $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MFOL}}_{\Sigma,\mathcal{M},\nu} r(t_1, \ldots, t_n)$ iff

$$(\nu(t_1), \ldots, \nu(t_n)) \in r^{\mathcal{M}}$$

- $\forall \varphi \in \mathcal{S}en_{\text{MFOL}}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MFOL}}_{\Sigma,\mathcal{M},\nu} \neg\varphi$ iff $(\mathfrak{E}, \mathfrak{R}) \not\models^{\text{MFOL}}_{\Sigma,\mathcal{M},\nu} \varphi$,
- $\forall x \in \mathcal{X}, \forall \varphi \in \mathcal{S}en_{\text{MFOL}}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MFOL}}_{\Sigma,\mathcal{M},\nu} (\forall x)\varphi$ iff $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MFOL}}_{\Sigma,\mathcal{M},\nu'} \varphi$ for every interpretation ν' such that $\nu'(y) = \nu(y)$ for every variable $y \in \mathcal{X}$, except maybe for x,
- $\forall \varphi \in \mathcal{S}en_{\text{MFOL}}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MFOL}}_{\Sigma,\mathcal{M},\nu} \Box\varphi$ iff $(\mathfrak{E}, \mathfrak{R}) \not\models^{\text{MFOL}}_{\Sigma,\mathcal{M}',\nu} \varphi$ for every state $\mathcal{M}' \in \mathfrak{E}$ such that $\mathcal{M}\mathfrak{R}\mathcal{M}'$,
- $\forall \varphi, \psi \in \mathcal{S}en_{\text{MFOL}}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MFOL}}_{\Sigma,\mathcal{M},\nu} \varphi \vee \psi$ iff $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MFOL}}_{\Sigma,\mathcal{M},\nu} \varphi$ or $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MFOL}}_{\Sigma,\mathcal{M},\nu} \psi$

2 Elements-Based Semantics

It is clear from Examples 3 and Example 4 that the satisfaction relation between formulae and models of a given first order signature is first defined with respect to the notion of interpretation of the variables. This is, in fact, a general case when dealing with logics whose semantics relies upon an interpretation of variables.

We will first show how it is possible by a simple trick on the syntax to simulate this concept of interpretation of variables while strictly staying in the limit of Institution Theory. This will then allow us to build, from a given institution, the associated institution of open formulae. We will then generalize the approach, introducing the framework of $\mathfrak{EBS}$-Institutions, in order to capture satisfaction based on states as well.

2.1 Internal Elements-Based Semantics of an Institution

Intuitively, the concept of formula satisfaction by a model of the same given signature is based on the concept of "internal satisfaction" of open formulae which is parametrized by the abstract valuation of the variables, *i.e.* a value is given to the variables. The formula satisfaction is then layered by this concept of internal satisfaction. In order to model this layered semantics we now introduce the concept of internal elements-based semantics ($i\mathcal{EBS}$).

Definition 2 (Internal Elements-Based Semantics). *Let* $\mathcal{I} = (\mathcal{S}ig, \mathcal{S}en, \mathcal{M}od, \models)$ *be an institution. An* internal elements-based semantics *of* $\mathcal{I}$ *is a 4-tuple* $\mathcal{I}^{ebs} = (\mathcal{S}ig^{ebs}, \mathcal{S}en^{ebs}, \mathcal{M}od^{ebs}, [\![]\!]^{ebs})$ *such that:*

- $\mathcal{S}ig^{ebs}$ *is a* category *such that:*
  - *objects are a certain* class of signature morphisms $\chi : \Sigma \to \Sigma'$ *of* $\mathcal{I}$*;*
  - *morphisms* $\sigma : (\chi_1 : \Sigma_1 \to \Sigma_1') \to (\chi_2 : \Sigma_2 \to \Sigma_2')$ *are* ordered pairs of signature morphisms $(\sigma_1 : \Sigma_1 \to \Sigma_2, \sigma_1' : \Sigma_1' \to \Sigma_2')$ *of* $\mathcal{I}$ *such that the*

following diagram is a weak amalgamation diagram.[4]

$$\begin{array}{ccc} \Sigma_1 & \xrightarrow{\chi_1} & \Sigma_1' \\ {\scriptstyle\sigma_1}\downarrow & & \downarrow{\scriptstyle\sigma_1'} \\ \Sigma_2 & \xrightarrow[\chi_2]{} & \Sigma_2' \end{array}$$

- $\mathcal{S}en^{ebs} : \mathcal{S}ig^{ebs} \to \mathcal{S}et$ *is a* functor *that associates every signature* $\chi : \Sigma \to \Sigma' \in |\mathcal{S}ig^{ebs}|$ *with the set* $\mathcal{S}en(\Sigma') \in |\mathcal{S}et|$ *of* Σ'-sentences *of* $\mathcal{I}$*;*
- $\mathcal{M}od^{ebs} : \mathcal{S}ig^{ebs} \to \mathcal{C}at^{op}$ *is a* functor *that associates every* $\chi : \Sigma \to \Sigma' \in |\mathcal{S}ig^{ebs}|$ *with the category* $\mathcal{M}od(\Sigma) \in |\mathcal{C}at|$ *of* Σ-models *of* $\mathcal{I}$*;*
- $[\![]\!]^{ebs} = ([\![]\!]^{ebs}_\chi)_{\chi \in |\mathcal{S}ig^{ebs}|}$ *is a* family of functors *such that for every signature* $\chi \in |\mathcal{S}ig^{ebs}|$*, the functor* $[\![]\!]^{ebs}_\chi : \mathcal{M}od^{ebs}(\chi) \to \mathcal{S}et$ *associates every model* $\mathcal{M} \in |\mathcal{M}od^{ebs}(\chi)|$ *with the set* $[\![\mathcal{M}]\!]^{ebs}_\chi = \{\mathcal{M}' \in |\mathcal{M}od(\Sigma')| / \mathcal{M}od(\chi)(\mathcal{M}') = \mathcal{M}\}$ *of* states of $\mathcal{M}$*.*

For every χ*-model* $\mathcal{M} \in |\mathcal{M}od^{ebs}(\chi)|$ *and every* χ*-sentence* $\varphi' \in \mathcal{S}en^{ebs}(\chi)$*, one defines the* satisfaction of φ' by $\mathcal{M}$ at state $\mathcal{M}' \in [\![\mathcal{M}]\!]^{ebs}_\chi$ *in the following way:*

$$\mathcal{M} \models_{\chi,\mathcal{M}'} \varphi' \Leftrightarrow \mathcal{M}' \models_{\Sigma'} \varphi'$$

Finally, a χ-model $\mathcal{M} \in |\mathcal{M}od^{ebs}(\chi)|$ satisfies a χ-sentence $\varphi' \in \mathcal{S}en^{ebs}(\chi)$*, denoted* $\mathcal{M} \models_\chi \varphi'$*, if and only if for every state* $\mathcal{M}' \in [\![\mathcal{M}]\!]^{ebs}_\chi$ *one has* $\mathcal{M} \models_{\chi,\mathcal{M}'} \varphi'$*.*

Intuitively, the above definition indicates that the concept of variable can be encoded syntactically into the signatures. Indeed, the domain of interpretation of variables (no matter what their nature is) is the same as the domain of interpretation of the elements of the signatures. Ultimately, a variable stands for an element of the signature, only this element is not specified (apart from its nature). So a signature with variables is nothing but a signature with distinguished (and possibly empty) subsets of sort signs, function signs and relation signs. This clearly defines a specific class of signature morphisms (the objects of the category $\mathcal{S}ig^{ebs}$) depending on the kind of variables one wants to consider.

Example 5. The internal elements-based semantics of FOL is defined in the following way:

- $\mathcal{S}ig^{ebs}_{\mathrm{FOL}}$ is the category objects of which are the signature morphisms $\chi : (\mathcal{F}, \mathcal{R}) \to (\mathcal{F} \cup \mathcal{X}, \mathcal{R})$ where $\mathcal{X}$ is the set of variable associated with $(\mathcal{F}, \mathcal{R})$;[5]
- $\mathcal{S}en^{ebs}_{\mathrm{FOL}} : \mathcal{S}ig^{ebs}_{\mathrm{FOL}} \to \mathcal{S}et$ is the functor that associates every signature $\chi : (\mathcal{F}, \mathcal{R}) \to (\mathcal{F} \cup \mathcal{X}, \mathcal{R})$ with the set of first order $(\mathcal{F} \cup \mathcal{X}, \mathcal{R})$-formulae (both closed and open);

[4] This means that for every Σ_1-model $\mathcal{M}_1 \in |\mathcal{M}od(\Sigma_1)|$ and every Σ_2-model $\mathcal{M}_2 \in |\mathcal{M}od(\Sigma_2)|$ such that $\mathcal{M}od(\sigma_1)(\mathcal{M}_1) = \mathcal{M}od(\sigma_2)(\mathcal{M}_2)$, there exists a Σ'-model $\mathcal{M}' \in |\mathcal{M}od(\Sigma')|$ such that $\mathcal{M}od(\sigma_1')(\mathcal{M}') = \mathcal{M}_1$ and $\mathcal{M}od(\sigma_2')(\mathcal{M}') = \mathcal{M}_2$.

[5] This is the class of representable morphisms introduced in [Dia05].

- $\mathcal{M}od^{ebs}_{\mathrm{FOL}} : \mathcal{S}ig^{ebs}_{\mathrm{FOL}} \rightarrow \mathcal{C}at^{op}$ is the functor that associates every signature $\chi : (\mathcal{F}, \mathcal{R}) \rightarrow (\mathcal{F} \cup \mathcal{X}, \mathcal{R})$ with the category of $(\mathcal{F}, \mathcal{R})$-models;
- For every signature $\chi : (\mathcal{F}, \mathcal{R}) \rightarrow (\mathcal{F} \cup \mathcal{X}, \mathcal{R})$ and every $(\mathcal{F}, \mathcal{R})$-model, $[\![\mathcal{M}]\!]_\chi = M^{\mathcal{X}}$, *i.e.* the set of interpretation of variables $\nu : \mathcal{X} \rightarrow M$.

Thus the states associated with a given model are the interpretations of variables in this model. Indeed, an interpretation of first order variables is nothing but a value $m \in M$ assigned to every variable $x \in \mathcal{X}$. One can see a mapping from $\mathcal{X}$ to M as a $(\mathcal{F} \cup \mathcal{X}, \mathcal{R})$-extension of $\mathcal{M}$.

Satisfaction in purely modal logics (*i.e.* based on propositional logic) does only depends on the states. These states being implicit, it is not possible to define an internal elements-based semantics for modal propositional logic. Indeed, in order to do so one would have to add to the signatures elements to represent the states and define a distinguished class of signature morphisms.

What about modal logics that are not defined as an extension of propositional logic such as modal first order logic? Having a closer look at it, it appears that there are in fact two layers in the definition of the satisfaction relation, one of them being a first order one (*cf.* Ex. 4). So, this first order layer can be captured by the concept of internal elements-based semantics. One then talks of partial internal elements-based semantics of modal logic with variables. This elements-based semantics is only visible to the one who is interested on how satisfaction is done inside every state of a given model. For lack of space we do not show this construction here.

Remark 1. Propositional Logic is a neutral element of the internal elements-based semantics construction. Although its semantics relies on a certain notion of interpretation of variables (the valuation of propositional variables), the satisfaction relation is defined in one step, *i.e.* no semantical structure is pre-supposed in which several different interpretations of the variables can be given. In fact, a semantical structure is a valuation of propositional variables.

The interest of the concept of internal elements-based semantics is that it allows to take into account the concept of open formula within institution theory. As free variables and open formulae are two very important concepts of Classical Model Theory a certain number of constructions and results relying on these two concepts could be generalized at the level of Institution Theory using internal elements-based semantics.

Proposition 1. *The internal elements-based semantics $\mathcal{I}^{ebs}$ of an institution $\mathcal{I}$ is itself an institution.*

2.2 $\mathfrak{EBS}$-Institutions

Internal elements-based semantics is nothing but an encoding in the syntax of those elements used to define the satisfaction relation. But satisfaction in temporal/modal logics is based on the concept of state/world. This is a semantical concept and it cannot be encoded in the syntax without altering it.

In order to solve this problem, the framework of $\mathfrak{EBS}$-institution has been introduced in [Bar05].[6] This framework allows to model in a very abstract way the elements used to define formula satisfaction, *i.e.* independently of their nature. It refines the institutional framework by considering a certain set of states associated with every model. No specific information on the nature of these states is provided. Only the truth values of formulae in these states are defined. Satisfaction is then defined in a way much similar to satisfaction in Kripke semantics.

Definition 3 ($\mathfrak{EBS}$ pre-Institution). *A* $\mathfrak{EBS}$ pre-institution $\mathcal{I}^{\mathfrak{ebs}}$ *is a 4-tuple* $(\mathcal{Sig}, \mathcal{Sen}, \mathcal{Mod}, [\![]\!])$ *such that:*

- *The 3-tuple* $(\mathcal{Sig}, \mathcal{Sen}, \mathcal{Mod})$ *is defined as in the institutional case (cf. Def. 1);*
- $[\![]\!] = ([\![]\!]_\Sigma)_{\Sigma \in |\mathcal{Sig}|}$ *is a* family of functors *indexed by* $|\mathcal{Sig}|$ *such that for every signature* $\Sigma \in |\mathcal{Sig}|$, $[\![]\!]_\Sigma : \mathcal{Mod}(\Sigma) \to \mathcal{S}et$ *associates every* Σ*-model* $\mathcal{M} \in |\mathcal{Mod}(\Sigma)|$ *with a set of* states.

For every signature $\Sigma \in |\mathcal{Sig}|$, *every* Σ*-model* $\mathcal{M} \in |\mathcal{Mod}(\Sigma)|$ *and every* Σ*-sentence* $\varphi \in \mathcal{Sen}(\Sigma)$, *one defines the* satisfaction of φ by $\mathcal{M}$ *in the following way:*

$$\mathcal{M} \models_\Sigma \varphi \iff \forall \eta \in [\![\mathcal{M}]\!]_\Sigma,\ \mathcal{M} \models_{\Sigma,\eta} \varphi$$

where $\mathcal{M} \models_{\Sigma,\eta} \_ \subseteq \mathcal{Sen}(\Sigma)$ *is a unary relation defined over* $\mathcal{Sen}(\Sigma)$.

Example 6 (Propositional Logic). The $\mathfrak{EBS}$ pre-institution of propositional logic is defined as follows:

- $\mathcal{Sig}, \mathcal{Sen}, \mathcal{Mod}$ are defined as in Example 1;
- Given a signature Σ and a Σ-model $v : \Sigma \to \{0,1\}$, $[\![v]\!]_\Sigma = \mathbb{1}$, where $\mathbb{1}$ is a set with only one element. The relation $v \models_{\Sigma,\mathbb{1}}$ is defined as follows:
  - $\forall p \in \Sigma$, $v \models_{\Sigma,\mathbb{1}} p$ if and only if $v(p) = 1$,
  - $\forall \varphi \in \mathcal{Sen}(\Sigma)$, $v \models_{\Sigma,\mathbb{1}} \neg\varphi$ if and only if $v \not\models_{\Sigma,\mathbb{1}} \varphi$,
  - $\forall \varphi, \psi \in \mathcal{Sen}(\Sigma)$, $v \models_{\Sigma,\mathbb{1}} \varphi \vee \psi$ if and only if $v \models_{\Sigma,\mathbb{1}} \varphi$ or $v \models_{\Sigma,\mathbb{1}} \psi$.

It is then clear that $v \in |\mathcal{Mod}(\Sigma)|$, $v \models^{\text{PL}}_\Sigma \varphi$ if and only if $v \models_{\Sigma,\mathbb{1}} \varphi$.

Example 7 (Modal Propositional Logic). The $\mathfrak{EBS}$ pre-institution of modal propositional logic is defined as follows:

- $\mathcal{Sig}, \mathcal{Sen}, \mathcal{Mod}$ are defined as in Example 2;
- Given a signature Σ and a Σ-model $(\mathfrak{E}, \mathfrak{R})$, $[\![(\mathfrak{E}, \mathfrak{R}]\!]_\Sigma = \mathfrak{E}$ and, for every $\eta \in \mathfrak{E}$, $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta}$ is defined as follows:
  - $\forall p \in \Sigma$, $(\mathfrak{E}, \mathfrak{R}, \mathfrak{v}) \models_{\Sigma,\eta} p$ if and only if $\mathfrak{v}(\eta, p) = 1$,
  - $\forall \varphi \in \mathcal{Sen}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}, \mathfrak{v}) \models_{\Sigma,\eta} \neg\varphi$ if and only if $(\mathfrak{E}, \mathfrak{R}, \mathfrak{v}) \not\models_{\Sigma,\eta} \varphi$,
  - $\forall \varphi \in \mathcal{Sen}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}, \mathfrak{v}) \models_{\Sigma,\eta} \Box\varphi$ if and only if for all $\eta' \in \mathfrak{E}$ such that $\eta \mathfrak{R} \eta'$, one has $(\mathfrak{E}, \mathfrak{R}, \mathfrak{v}) \models_{\Sigma,\eta'} \varphi$,
  - $\forall \varphi, \psi \in \mathcal{Sen}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}, \mathfrak{v}) \models_{\Sigma,\eta} \varphi \vee \psi$ if and only if $(\mathfrak{E}, \mathfrak{R}, \mathfrak{v}) \models_{\Sigma,\eta} \varphi$ or $(\mathfrak{E}, \mathfrak{R}, \mathfrak{v}) \models_{\Sigma,\eta} \psi$.

[6] It was introduced under the term of "Stratified Institution". The author thinks that "Elements-Based Institution" is both more adequate and intuitive.

It is then clear that $v \models^{\text{MPL}}_{\Sigma} \varphi$ if and only if $v \models_{\Sigma,\eta} \varphi$ for every $\eta \in [\![(\mathfrak{E}, \mathfrak{R})]\!]_{\Sigma}$.

Example 8 (First Order Logic). The $\mathfrak{EBS}$ pre-institution of first order logic is defined as follows:

- $\mathcal{Sig}$ and $\mathcal{Mod}$ are defined as in Example 3;
- $\mathcal{Sen} : \mathcal{Sig} \rightarrow \mathcal{Set}$ is a functor that associates every signature $\Sigma \in |\mathcal{Sig}|$ with the set of all first order formulae built over Σ (*i.e.* including open formulae);
- Given a signature $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{R})$, a set of variables $\mathcal{X}$ and a Σ-structure $\mathcal{M}$, $[\![\mathcal{M}]\!]_{\Sigma} = M^{\mathcal{X}}$ and, for every interpretation of variables $\nu \in M^{\mathcal{X}}$, $\mathcal{M} \models_{\Sigma,\nu}$ is defined as follows:
  - $\forall t, t' \in T_{\Sigma}(\mathcal{X})$, $\mathcal{M} \models_{\Sigma,\nu} t = t'$ if and only if $\nu(t) = \nu(t')$,
  - $\forall r \in \mathcal{R}^n, \forall t_1, \ldots, t_n \in T_{\Sigma}(\mathcal{X})$, $\mathcal{M} \models_{\Sigma,\nu} r(t_1, \ldots, t_n)$ if and only if
  $$(\nu(t_1), \ldots, \nu(t_n)) \in r^{\mathcal{M}}$$
  - $\forall \varphi \in \mathcal{Sen}(\Sigma)$, $\mathcal{M} \models_{\Sigma,\nu} \neg\varphi$ if and only if $\mathcal{M} \not\models_{\Sigma,\nu} \varphi$,
  - $\forall x \in \mathcal{X}, \forall \varphi \in \mathcal{Sen}(\Sigma)$, $\mathcal{M} \models_{\Sigma,\nu} (\forall x)\varphi$ if and only if $\mathcal{M} \models_{\Sigma,\nu'} \varphi$ for every interpretation ν' such that $\nu'(y) = \nu(y)$ for every variable $y \in \mathcal{X}$, except eventually for x,
  - $\forall \varphi, \psi \in \mathcal{Sen}(\Sigma)$, $\mathcal{M} \models_{\Sigma,\nu} \varphi \vee \psi$ if and only if $\mathcal{M} \models_{\Sigma,\nu} \varphi$ or $\mathcal{M} \models_{\Sigma,\nu} \psi$

It is then clear that $\mathcal{M} \models^{\text{FOL}}_{\Sigma} \varphi$ if and only if $\mathcal{M} \models_{\Sigma,\eta} \varphi$ for every $\eta \in [\![\mathcal{M}]\!]_{\Sigma}$.

Example 9 (Modal First Order Logic). The $\mathfrak{EBS}$ pre-institution of modal first order logic is defined as follows:

- $\mathcal{Sig}, \mathcal{Sen}, \mathcal{Mod}$ are defined as in Example 4;
- Given a signature $\Sigma = (\mathcal{F}, \mathcal{R})$ and a Σ-model $(\mathfrak{E}, \mathfrak{R})$, $[\![(\mathfrak{E}, \mathfrak{R})]\!]_{\Sigma} = \mathfrak{E}$, for all $\eta \in \mathfrak{E}$, $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta}$ is defined inductively on the structure of formulae in the following way:
  - $\forall t, t' \in T_{\Sigma}(\mathcal{X})$, $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta} t = t'$ iff $\eta \models_{\Sigma} t = t'$,
  - $\forall r \in \mathcal{R}, \forall t_1, \ldots, t_n \in T_{\Sigma}(\mathcal{X})$, $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta} r(t_1, \ldots, t_n)$ ssi $\eta \models_{\Sigma} r(t_1, \ldots, t_n)$,
  - $\forall \varphi \in \mathcal{Sen}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta} \neg\varphi$ iff $(\mathfrak{E}, \mathfrak{R}) \not\models_{\Sigma,\eta} \varphi$,
  - $\forall x \in \mathcal{X}, \forall \varphi \in \mathcal{Sen}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta} (\forall x)\varphi$ iff $\eta \models_{\Sigma} (\forall x)\varphi$,
  - $\forall \varphi \in \mathcal{Sen}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta} \Box\varphi$ iff for all $\eta' \in \mathfrak{E}$ such that $\eta \mathfrak{R} \eta'$, we have $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta'} \varphi$,
  - $\forall \varphi, \psi \in \mathcal{Sen}(\Sigma)$, $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta} \varphi \vee \psi$ iff $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta} \varphi$ or $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta} \psi$.

It is then clear that $(\mathfrak{E}, \mathfrak{R}) \models^{\text{MFOL}}_{\Sigma} \varphi$ if and only if $(\mathfrak{E}, \mathfrak{R}) \models_{\Sigma,\eta} \varphi$ for every $\eta \in [\![(\mathfrak{E}, \mathfrak{R})]\!]_{\Sigma}$. In order to see the second level of this elements-based semantics, *i.e.* the one due to the interpretation of the variables, one must detail the expression $\eta \models_{\Sigma} \varphi$, where η is a first order model. Basically the work is the same than for Example 8

Definition 4 ($\mathfrak{EBS}$-Institution). *An* $\mathfrak{EBS}$-institution $\mathcal{I}^{\mathfrak{ebs}}$ *is a* $\mathfrak{EBS}$ *pre-institution that satisfies the satisfaction condition, i.e. :*
$\forall \sigma \in \mathcal{Sig}(\Sigma, \Sigma'), \forall \mathcal{M}' \in |\mathcal{Mod}(\Sigma')|, \forall \varphi \in \mathcal{Sen}(\Sigma)$,

$$\mathcal{M}' \models_{\Sigma'} \mathcal{Sen}(\sigma)(\varphi) \Leftrightarrow \mathcal{Mod}(\sigma)(\mathcal{M}') \models_{\Sigma} \varphi$$

Under a very simple hypothesis $\mathfrak{EBS}$ pre-institution present $\mathfrak{EBS}$ institutions.

Proposition 2. *Every* $\mathfrak{EBS}$ *pre-institution* $\mathcal{I}^{\mathfrak{ebs}} = (\mathcal{Sig}, \mathcal{Sen}, \mathcal{Mod}, [\![]\!])$ *such that for every signature morphism* $\sigma \in \mathcal{Sig}(\Sigma, \Sigma')$ *and every* Σ'*-model* $\mathcal{M}' \in |\mathcal{Mod}(\Sigma')|$ *there exists a natural transformation* $[\![]\!]_\sigma : [\![]\!]_{\Sigma'} \Rightarrow [\![]\!]_\Sigma \circ \mathcal{Mod}(\sigma)$ *that satisfies both following conditions:*

1. *for every* Σ'*-model* $\mathcal{M}' \in |\mathcal{Mod}(\Sigma')|$, $[\![\mathcal{M}']\!]_\sigma$ *is a surjective mapping;*
2. *for every* Σ*-sentence* $\varphi \in \mathcal{Sen}(\Sigma)$ *and every* $\eta' \in [\![\mathcal{M}']\!]_{\Sigma'}$*:*

$$\mathcal{M}' \models_{\Sigma', \eta'} \mathcal{Sen}(\sigma)(\varphi) \iff \mathcal{Mod}(\sigma)(\mathcal{M}') \models_{\Sigma, [\![\mathcal{M}']\!]_\sigma(\eta')} \varphi$$

is a $\mathfrak{EBS}$*-institution.*

The overall interest of $\mathfrak{EBS}$-institutions compared to the usual institutions is to be able to take into account open formulae. Indeed, the definition of satisfaction in the institutions restricts *de facto* the set of formulae of a given signature to the set of closed formulae (sentences) over this signature. But free variables and open formulae are two very important concepts of Model Theory. A certain number of constructions and results rely on these two concepts. As an example, the Method of Diagrams which allows to build elementary equivalent models is defined on the concept of free variables.

Because free variables and open formulae can be modelled using an elements-based semantics we believe that this construction is a powerful tool to investigate Institution-Independant Model Theory. Some results, such as the Method of Diagram (*cf.* [Bar05]) and the concept of Elementary Morphism (*cf.* [AD06]) have already been generalized at the institution level using an elements-based semantics.

The approach of elements-based semantics to Institution-Independant Model Theory is quite different in nature than the one followed by R. Diaconescu in [Dia05]. This last approach focuses on the notion of axiomatisation of models and uses it to investigate Model Theory at the institution level. We rather focus on the elements used to define the satisfaction relation. This gives a different and hopefully complementary view on Institution-Independant Model Theory.

3 Abstract Modalities and Connectors

In order to model the quantifiers and the modalities we now need to extend the elements-based semantics construction with a notion of abstract syntax. Although this extension is conceptually close to the approach followed with the parchments [MTP98], they are quite different in nature in the sense that we simply consider a set of atomic elements on which operate the elements of a set of operations. These two sets can be seen as the set of generators and the set of relations of an abstract group. In other words, the atomic formulae of the language are the generators and the connectives, quantifiers and modalities are the operations.

Before we can do so, we need to introduce the following notions.

Definition 5 (Constructor Signature). *A* constructor signature *is a family* $\mathcal{C} = (\mathcal{C}^n)_{n\in\mathbb{N}}$ *of sets indexed by the set of natural numbers. For every* $n \in \mathbb{N}$*, an element* $c \in \mathcal{C}^n$ *is called* constructor of arity n*. We will note either* $c \in \mathcal{C}^n$ *or* $c^n \in \mathcal{C}$*.*

Definition 6 (Constructor Morphism). *A* constructor morphism *is a mapping* $\varsigma : \mathcal{C}_1 \to \mathcal{C}_2$ *that preserves arity, i.e. :*

$$\forall n \in \mathbb{N}, \forall c \in \mathcal{C}_1^n,\ \varsigma(c) \in \mathcal{C}_2^n$$

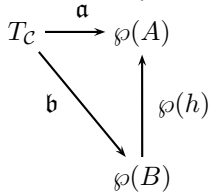
It is clear that constructor signatures are the objects of a category $\mathcal{LogSig}$, morphisms of which are the constructor morphisms.

Notation 31 *Let* $\mathcal{C} = (\mathcal{C}^n)_{n\in\mathbb{N}}$ *be a constructor signature. We will note* $T_{\mathcal{C}}$ *the set of terms built over the signature* $\mathcal{C}$*.*

Definition 7 (Constructor Algebra). *Let* $\mathcal{C} = (\mathcal{C}^n)_{n\in\mathbb{N}}$ *be a constructor signature. A* $\mathcal{C}$-algebra $\mathfrak{A}$ *is a carrier set* A *together with a mapping* $\mathfrak{a} : T_{\mathcal{C}} \to \wp(A)$*.*

The elements of the carrier set of a $\mathcal{C}$-algebra are called *states* in the following.

Definition 8 ($\mathcal{C}$-Algebra Morphism). *Let* $\mathcal{C} = (\mathcal{C}^n)_{n\in\mathbb{N}}$ *be a constructor signature and let* $\mathfrak{A}$ *and* $\mathfrak{B}$ *be two* $\mathcal{C}$*-algebras. A* morphism of $\mathcal{C}$-algebras *from* $\mathfrak{A}$ *to* $\mathfrak{B}$ *is a mapping* $h : A \to B$ *such that the following diagram is commutative:*



It is clear that for every constructor signature $\mathcal{C} \in |\mathcal{LogSig}|$ the $\mathcal{C}$-algebras are the objects of a category $\mathcal{LogMod}$, morphisms of which are the $\mathcal{C}$-algebras morphisms. This allows us to define the contravariant functor $\mathcal{LogMod} : \mathcal{LogSig} \to \mathcal{C}at$ that associates with every constructor signature $\mathcal{C} \in |\mathcal{LogSig}|$ the category of $\mathcal{C}$-models $\mathcal{LogMod}(\mathcal{C})$ and, to every constructor morphism $\varsigma \in \mathcal{LogSig}(\mathcal{C}, \mathcal{C}')$, the forgetful functor $\mathcal{LogMod}(\varsigma) : \mathcal{LogMod}(\mathcal{C}') \to \mathcal{LogMod}(\mathcal{C})$ such that for every $\mathcal{C}'$-algebra of constructor $\mathfrak{A}' \in |\mathcal{LogMod}(\mathcal{C}')|$, $\mathcal{LogMod}(\varsigma)(\mathfrak{A}') = \mathfrak{A} \in |\mathcal{LogMod}(\mathcal{C})|$ is the $\mathcal{C}$-algebra defined by $A = \mathcal{LogMod}(\varsigma)(A')$ and, for every constructor symbol $c \in \mathcal{C}$, $\mathfrak{a}(c) = \mathfrak{a}'(\varsigma(c))$.

The previous definition shows that for every constructor signature $\mathcal{C} \in |\mathcal{LogSig}|$, the $\mathcal{C}$-algebras associate with a set of states every formula built over $\mathcal{C}$ (or $\mathcal{C}$-term). Intuitively this set is the set of states for which the formula is true. The satisfaction of a formula $\varphi \in T_{\mathcal{C}}$ in a $\mathcal{C}$-algebra $\mathfrak{A} \in |\mathcal{LogMod}(\mathcal{C})|$ is defined in the same way:

$$\mathfrak{A} \models_{\mathcal{C}} \varphi \Leftrightarrow \forall \eta \in A,\ \mathfrak{A} \models_{\mathcal{C},\eta} \varphi$$

where $\mathfrak{A} \models_{\mathcal{C},\eta} \varphi \Leftrightarrow \eta \in \mathfrak{a}(\varphi)$.

Notation 32 *In order to use lighter notations we will note* $[\![]\!]_{\mathcal{C}}$ *the mapping that associates with every* $\mathcal{C}$*-algebra* $\mathfrak{A} = (A, \mathfrak{a})$ *its carrier set* A*. The* $\mathcal{C}$*-algebra* $\mathfrak{A} = (A, \mathfrak{a})$ *can then be noted* $([\![A]\!], A)$ *without any ambiguity.*

We can now make the link with the elements-based semantics.

Definition 9 ($\kappa\mathfrak{EBS}$-institution). *A* $\kappa\mathfrak{EBS}$-institution $\kappa\mathcal{I}^{\mathfrak{ebs}}$ *is the providing of an* $\mathfrak{EBS}$*-institution* $\mathcal{I}^{\mathfrak{ebs}} = (\mathcal{Sig}, \mathcal{Sen}, \mathcal{Mod}, [\![]\!])$ *and a functor* $\kappa : \mathcal{Sig} \to \mathcal{LogSig}$ *such that for every signature* $\Sigma \in |\mathcal{Sig}|$ *there exists a function* $\beta_\Sigma : |\mathcal{Mod}(\Sigma)| \to |\mathcal{LogMod}(\kappa(\Sigma))|$ *natural in* Σ *(cf. Fig. 1) and:*

- $\forall \Sigma \in |\mathcal{Sig}|$, $\mathcal{Sen}(\Sigma) = T_{\kappa(\Sigma)}$;
- $[\![\mathcal{M}]\!]_\Sigma = [\![\beta_\Sigma(\mathcal{M})]\!]_{\kappa(\Sigma)}$ *for every* Σ*-model* $\mathcal{M} \in |\mathcal{Mod}(\Sigma)|$;
- $\mathcal{M} \models_{\Sigma,\eta} \varphi$ *iff* $\beta_\Sigma(\mathcal{M}) \models_{\kappa(\Sigma),\eta} \varphi$.

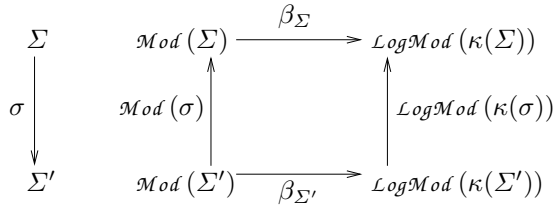


Fig. 1. Σ-models and $\kappa(\Sigma)$-algebras

The previous definition show that for every signature $\Sigma \in |\mathcal{Sig}|$, the satisfaction of a Σ-formula $\varphi \in \mathcal{Sen}(\Sigma)$ in a Σ-model $\mathcal{M} \in |\mathcal{Mod}(\Sigma)|$ depends on the set of states associated with φ in the corresponding $\kappa(\Sigma)$-algebra $\beta_\Sigma(\mathcal{M})$. One will also note that the functor $\mathcal{Sen}$ is now redundant with functor κ. In the following we will present a $\kappa\mathfrak{EBS}$-institution as a 4-tuple $(\mathcal{Sig}, \kappa, \mathcal{Mod}, [\![]\!])$ when the existence of the associated $\mathfrak{EBS}$-institution will not be pre-supposed.

Example 10 (Propositional Logic). The $\kappa\mathfrak{EBS}$-institution of propositional logic is the providing of the couple $(\mathcal{I}^{\mathfrak{ebs}}_{\text{prop}}, \kappa_{\text{prop}})$ such that $\mathcal{I}^{\mathfrak{ebs}}_{\text{prop}}$ is the $\mathfrak{EBS}$-institution defined in Example 6 and $\kappa_{\text{prop}} : \mathcal{Sig}_{\text{prop}} \to \mathcal{LogSig}$ is the functor such that for every signature $\Sigma \in |\mathcal{Sig}_{\text{prop}}|$, $\kappa_{\text{prop}}(\Sigma)$ is the signature $(\mathcal{C}^n)_{n\in\mathbb{N}}$ of constructors defined as:

- $\mathcal{C}^0 = \Sigma$;
- $\mathcal{C}^1 = \{\neg\}$;
- $\mathcal{C}^2 = \{\vee\}$;
- $\forall n > 2$, $\mathcal{C}^n = \varnothing$.

Moreover, $\beta_\Sigma : |\mathcal{Mod}_{\text{prop}}(\Sigma)| \to |\mathcal{LogMod}(\kappa_{\text{prop}}(\Sigma))|$ is the function such that for every Σ-model $v \in |2^\Sigma|$, $\beta_\Sigma(v) = (\mathbb{1}, \mathfrak{a})$ where $\mathfrak{a} : T_{\kappa_{\text{prop}}(\Sigma)} \to \wp(\mathbb{1})$ is defined inductively on the structure of formulae in the following way:

- $\forall \varphi \in \Sigma,\ \mathfrak{a}(\varphi) = \mathbb{1}$ if $v(\varphi) = 1$
  $\varnothing$ otherwise
- $\forall \varphi \in T_{\kappa_{\text{prop}}(\Sigma)}, \mathfrak{a}(\neg\varphi) = \mathbb{1} \setminus \mathfrak{a}(\varphi)$;
- $\forall \varphi, \psi \in T_{\kappa_{\text{prop}}(\Sigma)}, \mathfrak{a}(\varphi \vee \psi) = \mathfrak{a}(\varphi) \cup \mathfrak{a}(\psi)$.

Example 11 (Modal Propositional Logic). The $\kappa\mathfrak{EBS}$-institution of modal propositional logic is the providing of the couple $(\mathcal{I}^{\mathfrak{ebs}}_{\text{mod}}, \kappa_{\text{mod}})$ such that $\mathcal{I}^{\mathfrak{ebs}}_{\text{mod}}$ is the $\mathfrak{EBS}$-institution defined in the Example 7 and $\kappa_{\text{mod}} : \mathcal{Sig}_{\text{mod}} \rightarrow \mathcal{LogSig}$ is the functor such that for every signature $\Sigma \in |\mathcal{Sig}_{\text{mod}}|$, $\kappa_{\text{mod}}(\Sigma)$ is the signature $(\mathcal{C}^n)_{n\in\mathbb{N}}$ of constructors defined as:

- $\mathcal{C}^0 = \Sigma$;
- $\mathcal{C}^1 = \{\neg, \Box\}$;
- $\mathcal{C}^2 = \{\vee\}$;
- $\forall n > 2, \mathcal{C}^n = \varnothing$.

Moreover, $\beta_\Sigma : |\mathcal{Mod}_{\text{mod}}(\Sigma)| \rightarrow |\mathcal{LogMod}(\kappa_{\text{mod}}(\Sigma))|$ is the function such that for every Σ-model $(\mathfrak{E}, \mathfrak{R}) \in |\mathcal{Mod}_{\text{mod}}(\Sigma)|$, $\beta_\Sigma((\mathfrak{E}, \mathfrak{R})) = (\mathfrak{E}, \mathfrak{a})$ where $\mathfrak{a} : T_{\kappa_{\text{mod}}(\Sigma)} \rightarrow \wp(\mathfrak{E})$ is defined inductively on the structure of formulae in the following way:

- $\forall \varphi \in \Sigma, \mathfrak{a}(\varphi) = \{\eta \in \mathfrak{E} / \mathfrak{v}(\varphi, \eta) = 1\}$;
- $\forall \varphi \in T_{\kappa_{\text{mod}}(\Sigma)}, \mathfrak{a}(\neg\varphi) = \mathfrak{E} \setminus \mathfrak{a}(\varphi)$;
- $\forall \varphi, \psi \in T_{\kappa_{\text{mod}}(\Sigma)}, \mathfrak{a}(\varphi \vee \psi) = \mathfrak{a}(\varphi) \cup \mathfrak{a}(\psi)$.
- $\forall \varphi \in T_{\kappa_{\text{mod}}(\Sigma)}, \mathfrak{a}(\Box\varphi) = \mathfrak{R}^{-1}\mathfrak{a}(\varphi)$;

Example 12 (First Order Logic). The $\kappa\mathfrak{EBS}$-institution of first order logic is the providing of the couple $(\mathcal{I}^{\mathfrak{ebs}}_{\text{lpo}}, \kappa_{\text{lpo}})$ such that $\mathcal{I}^{\mathfrak{ebs}}_{\text{lpo}}$ is the $\mathfrak{EBS}$-institution defined in the Example 8 and $\kappa_{\text{lpo}} : \mathcal{Sig}_{\text{lpo}} \rightarrow \mathcal{LogSig}$ is the functor such that for every signature $\Sigma \in |\mathcal{Sig}_{\text{lpo}}|$, $\kappa_{\text{lpo}}(\Sigma)$ is the signature $(\mathcal{C}^n)_{n\in\mathbb{N}}$ of constructors defined as:

- $\mathcal{C}^0 = \mathcal{A}tom_\Sigma$;
- $\mathcal{C}^1 = \{\neg\} \cup \bigcup_{x\in\mathcal{X}}\{\forall x\}$;
- $\mathcal{C}^2 = \{\vee\}$;
- $\forall n > 2, \mathcal{C}^n = \varnothing$.

Moreover, $\beta_\Sigma : |\mathcal{Mod}_{\text{lpo}}(\Sigma)| \rightarrow |\mathcal{LogMod}(\kappa_{\text{lpo}}(\Sigma))|$ is the function such that for every Σ-model $\mathcal{M} \in |\mathcal{Mod}(\Sigma)|$, $\beta_\Sigma(\mathcal{M}) = (M^{\mathcal{X}}, \mathfrak{a})$ where $\mathfrak{a} : T_{\kappa_{\text{lpo}}(\Sigma)} \rightarrow \wp(M^{\mathcal{X}})$ is defined inductively on the structure of formulae in the following way:

- $\forall \varphi \in \mathcal{A}tom_\Sigma, \mathfrak{a}(\varphi) = \{\nu \in M^{\mathcal{X}} / \mathcal{M} \models_{\Sigma,\nu} \varphi\}$;
- $\forall \varphi \in T_{\kappa_{\text{lpo}}(\Sigma)}, \mathfrak{a}(\neg\varphi) = M^{\mathcal{X}} \setminus \mathfrak{a}(\varphi)$;
- $\forall \varphi, \psi \in T_{\kappa_{\text{lpo}}(\Sigma)}, \mathfrak{a}(\varphi \vee \psi) = \mathfrak{a}(\varphi) \cup \mathfrak{a}(\psi)$.
- $\forall \varphi \in T_{\kappa_{\text{lpo}}(\Sigma)}, \forall \nu \in M^{\mathcal{X}}$, $\nu \in \mathfrak{a}(\forall x\varphi)$ iff $\nu' \in \mathfrak{a}(\varphi)$ for every interpretation $\nu' \in M^{\mathcal{X}}$ such that $\nu'(y) = \nu(y)$ for every $y \in \mathcal{X} \setminus \{x\}$.

We now have a complete set of tools to reason about quantifiers and modalities. Not only we can investigate both notions of quantification and modality. and give a more abstract view on existing work on the subject, but we can also investigate the notion of logic combination using the power of the abstract syntax.

4 Future Work

We have introduced two constructions that allow to model the elements on which the definition of the satisfaction relation relies, namely internal elements-based semantics and (external) elements-based semantics. Although the internal elements-based semantics construction is quite limited we believed that it can be a useful tool to investigate predicate-like satisfaction. On the other hand the more general construction of elements-based semantics does not have these limitations. It seems to be a proper tool to investigate both notions of quantification and modality in a unified framework and at an abstract level. Moreover, as shown by some results already obtained ([Bar05],[AD06]), $\mathfrak{EBS}$-institutions allow investigations in Institution-Independent Model Theory and generalisation of Classical Model Theory results to the level of Institution Theory. This latter approach is quite different in nature from the one followed in [Dia05] as it focuses only on the definition of the satisfaction relation.

Future work include the following:

- Investigate both concepts of modality and quantification, their similarities and differences, as well as the methods and technics to introduce quantification and/or modalities in a logic or to temporalise a logic;
- Investigate the different existing semantics (possible world semantics, counterpart semantics, *etc.*) for modal and temporal logics using elements-based semantics;
- Investigate logic combination. At each steps of the validity check of a formula, and depending on the constructor to be interpreted, we would use the semantics that corresponds to this constructor, discarding all other elements of the original structure.

References

[AD06] Aiguier, M. and Diaconescu, R, *Stratified Institutions and Elementary Homomorphisms*, Information Processing Letters, Vol. 103 , 2007.

[Bar05] Barbier, F., *Résultats de théorie abstraite des modèles dans le cadre des institutions : vers la combinaison de logiques*, Université d'Évry-Val d'Éssonne, 2005.

[BG80] Burstall, R. and Goguen, J., *The Semantics of CLEAR, a Specification Language*, Proceedings of the Winter School on Abstract Software Specification (WSASS'80), LNCS Vol. 86, 1980.

[Dia05] Diaconescu, R., *Institution-Independant Model Theory, Birkhäuser*, 2008,

[GB92] Goguen, J. and Burstall, R., *Institutions: Abstract Model Theory for Specification and Programming*, Journal of the Association for Computing Machinery, 1992.

[MTP98] Mossakowski, T., Tarlecki, A. and Pawlowski, W., *Combining and Representing Logical Systems Using Model-Theoretic Parchments*, Recent Trends in Abstract Data Type, LNCS Vol. 1376, 1998.

Verifying Separation for a Monadic Scheduler

Tom Harke

Department of Computer Science, Portland State University, Portland OR, USA
harke@cs.pdx.edu

Abstract. In this paper we study separation proofs for schedulers written in a typed λ-calculus plus a monad. First we demonstrate separation proofs at a high level of reasoning for a variety of simple schedulers. Second, we lay the foundations for mechanizing these in the proof assistant Coq. We also argue why a proof of separation for a model extends to a proof of separation for code running on hardware.

1 Introduction

This paper demonstrates separation proofs at a high level of reasoning, for schedulers written in the λ-calculus plus a monad. It then lays the foundations for mechanizing these. The intent of this foundation is to allow proofs about more elaborate schedulers.

The separation problem is: given a scheduler managing many processes, does any one process have an undue influence upon another? If we focus on a particular process, called the *process of interest* (abbreviated POI) then an alternative phrasing of the problem is: when the POI runs interleaved with unrelated processes, does its behavior differ from when it runs alone? A more concrete phrasing depends on fixing parameters such as: the number of POIs, the specific scheduler, and whether the processes are non-deterministic. One instance of the problem is:

$$\mathrm{LR}\, d \to \varrho\ \langle c, d\rangle \cong \varrho\ \langle c\rangle$$

where ϱ is a round-robin scheduler, $\langle c, d\rangle$ and $\langle c\rangle$ are queues of processes, the precondition on d, similar in spirit to Rushby's local respect [Rus92], says that atomic events associated with d are ignored by the equivalence relation $\cong$.

The separation problem comprises two questions. First, whether the components of the scheduler are "correct," for an appropriate notion of correctness. For the memory system, one simple notion of correctness is that distinct processes access disjoint regions of memory. Second, assuming correct parts, whether the scheduler maintains separation. We refer to this latter question as *the Assembly Problem*.

This paper focuses on the Assembly Problem for various schedulers, on the proof techniques needed to solve it, and how to mechanize these proofs in Coq. There is little detail here about the ongoing mechanization in Coq, but Coq's limitations influenced this presentation.

The context for this work is the *House* operating system [HJLT05]. House is written in a strongly typed pure functional language, Haskell, plus the *H monad* (or *hardware monad*). The H monad is intended to be a small, coherent interface, a more easily formalized alternative to Haskell's `IO` monad. The type system rules out many sources of bugs, but it not strong enough to show separation.

1.1 Equivalences.

A central notion in the proofs is an equivalence relation between computations. The intuition behind it is that, if we could instrument the code to log memory accesses of the POI, we could detect an undue influence by a change in the logs. However, the presence of some accesses should have no effect: if $\mathrm{LR}\, d$, then work done by d should be ignored by the equivalence.

There are a number of notions of equivalence in this paper. In most cases we use $=$, e.g. to express laws. There are three cases where a more explicit equivalence is used. The first is definitional equality, $:=$, where the name on the left is defined by the expression on the right. A name in a definition may have parameters and may involve pattern matching, as is common in Haskell. The second is the equivalence on OS schedulings, $\cong$, which is biased to ignore some actions. The third, an arbitrary equivalence relation on type A is denoted by $\equiv_A$ (or $\equiv$ when the type is clear). It will turn out that $\cong$ is $\equiv_{\mathbb{M}A}$, for an appropriate monad $\mathbb{M}$.

1.2 Overview

The rest of the paper is as follows. §2 reviews background material: types, monads, sufficient completeness, and coinduction. §3 has a number of high level proofs of separation, assuming an adequate notion of equivalence. §4 discusses the preliminaries to mechanizing proofs: embedding a subset of Haskell into Coq, an applicative structure on setoids, details of a new monad used to implement that notion of equivalence, and a new monad used to allow non-terminating computation. §5 concludes.

2 Background

2.1 Types

Terms are classified by type with the '$:$' ('inhabits' or 'has type') relation. For instance $3 : \mathbb{N}$ asserts that 3 is a natural number.

There are a number of base types. Some are concrete: 1 is the type with one inhabitant, $()$; $\mathbb{N}$ is the usual type of natural numbers. Some are abstract: $\mathbb{P}$ is a type of process identifiers, with decidable equality; $\mathbb{C}$ is a type of process contexts, each of which has an associated process identifier accessed via $\mathrm{ID} : \mathbb{C} \to \mathbb{P}$, and each of which holds enough information to pause and resume a computation. Other operations on these abstract types are introduced as needed.

There are five type constructions: products, sums, arrows, and least and greatest fixed points. The first three are binary: given types A and B, the product, sum, and arrow are denoted by $A \times B$, $A + B$ and $A \to B$ respectively. Product terms are built by tupling, and taken apart with projection functions ($\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$). Sum terms are built with injection functions ($\iota_1 : A \to A + B$ and $\iota_2 : B \to A + B$) and taken apart with case analysis. One very useful sum is $Option\ A := 1 + A$, where $None$ is often used in place of $\iota_1\ ()$, and $Some\ a$ for $\iota_2\ a$. Arrow terms are built by λ-abstraction, and consumed by function application. We use the style of functional programmers, writing functions curried, instead of the fully applied style of mathematicians. That is, instead of $f(a, b)$ we write $f\ a\ b$.

The last two constructors are fixed points. They build recursive types by taking a functor F describing one level of data and repeating it. For instance, when $F\ X := 1 + \mathbb{N} \times X$, then $\mu X.1 + \mathbb{N} \times X$ is the type of finite lists of natural numbers, while $\nu X.1 + \mathbb{N} \times X$ contains both finite and infinite lists. Terms from either fixed point can be built with F, or decomposed with case analysis. The least fixed point $\mu X.F\ X$ creates inductive data, that is, terms built up by finitely many steps of F. The greatest fixed point $\nu X.F\ X$ creates co-inductive data, that is, terms which permit finitely many decomposition steps along the structure of F.

2.2 Monads

Monads allow the integration of effectful computations with a pure language by using a type distinction to isolate pure terms from impure ones, by restricting access to impure terms, and by requiring sequencing of impure terms. State, exceptions and nondeterminism are effects that monads can model. We will introduce two more monads in §4.3 and §4.4: the first adds the ability to define custom program equivalences, the basis for $\cong$, and the second allows Coq to naturally model non-termination.

Operations and Laws. A monad is a triple:

$$\begin{aligned} \mathbb{M} &: Type \to Type \\ \eta &: \forall A.A \to \mathbb{M}A \\ \star &: \forall A\ B.\mathbb{M}A \to (A \to \mathbb{M}B) \to \mathbb{M}B \end{aligned}$$

satisfying the laws:

$$\begin{aligned} \text{associativity:}\quad & (p \star q) \star r = p \star (\lambda a.qa \star r) \\ \text{left-identity:}\quad & \eta\ a \star q = qa \\ \text{right-identity:}\quad & p \star \eta = p \end{aligned}$$

$\mathbb{M}$ is a type function taking a *value* type to a *computation* type. For instance, if $\mathbb{N}$ is the type of natural number values, then $\mathbb{M}\mathbb{N}$ is the type of a computation that, when run, has *side effects* but ultimately produces a natural number. We'll

use the term *A-value* to denote a term from A, and *A-computation* to denote one from $\mathbb{M}A$. The polymorphic function η (read as 'return' or 'eta') has no effect: it simply promotes a value to a computation. The operator $\star$ ('bind') sequences two computations: the effects of the first occur before those of the second, and the value computed by the first seeds the second.

The operator $\gg$ ('sequence') is a specialization of $\star$ which doesn't pass a value:

$$\gg : \forall AB.\mathbb{M}A \to \mathbb{M}B \to \mathbb{M}B \qquad p \gg q := p \star \lambda\_.q$$

These operators are rough analogs to constructs in imperative languages. Bind is similar to semicolon (`;`), except for the value explicitly passed by the λ-binding. Sequence corresponds exactly to semicolon. Return is similar to an empty block `{}` in C, or `skip` in Algol, except for its value. The reason for the explicit value is that monads do not presume the presence of state; in contrast, in C the two subcomputation of '`;`' communicate implicitly by writing to and reading from state. Extending the analogy, the associativity law justifies why procedural abstraction is valid.

A wide variety of effects (and many combinations) can be modelled with monads. Each effect has associated operations and laws. For instance one interface for a state monad, which silently threads a component of type S, has operations *put* to set the state, *get* to retrieve the state within a computation, and *run* to execute the computation (Figure 1). There are additional laws relating *put*, *get*, *run*, $\star$ and η.

name	type
get	$\mathbb{M}S$
put	$S \to \mathbb{M}1$
run	$\mathbb{M}A \to S \to A \times S$

Fig. 1. State Monad Operations

$$\begin{array}{|ll|}
\hline
\text{run } (\eta\ a)\ s & = (a, s) \\
\text{run } (p \star q)\ s & = \text{uncurry } (\text{run } q)\ (\text{run } p\ s) \\
\text{run get } s & = (s, s) \\
\text{run } (\text{put } t)\ s & = ((), t) \\
\hline
\end{array}$$

Fig. 2. State Monad Axioms

There is another formulation of monads as a 4-tuple $(\mathbb{M}, \eta, join, map)$ where $join : \forall A.\mathbb{M}(\mathbb{M}A) \to \mathbb{M}A$ and $map : \forall A\ B.(A \to B) \to (\mathbb{M}A \to \mathbb{M}B)$ with laws relating the operators [Mac71]. While this alternative is more natural in many contexts, the first formulation is more natural for imperative programming. The two formulations are equivalent using:

$$\text{join}\, m = m \star id \qquad \text{map}\, f\ m = m \star (\eta \circ f) \qquad m \star k = \text{join}\ (\text{map}\, k\ m)$$

2.3 Sufficient Completeness

The H-interface is a common interface to both actual hardware and a model implemented in the proof assistant Coq. In order to claim that a property of the

latter holds in the former we need a means to show that the observable behavior of the model, with respect to the interface, is the same as hardware. To do so we view the H-monad as an *abstract data type* (or ADT), and provide a *sufficiently complete* axiomatization of it [TM92][Chapter 12].

Definition 2.1. *When a new type is defined as an ADT it has associated* operators*. Each operator is classified as either a* constructor*, if its return type is the new type, or an* accessor*, if its return type is an existing type.*

Definition 2.2. *An axiomatization of an ADT is* sufficiently complete *if every well-formed ground term built with that ADT's operators either has the new type, or is equivalent, via the axioms, to a term from an existing type.*

Given a sufficiently complete axiomatization of an ADT and two implementations, the behaviors observable via accessors are identical.

Queues. Queues are used both to implement round robin schedulers, and as an example of a sufficiently complete specification.

Let $\mathbb{Q}A$ denote the type of queues containing elements of type A. Figure 3 shows one possible set of queue operators: the four constructors are *empty* (abbreviated $\langle\rangle$), *enqueue* ($\lhd$), *append* ($+\!\!+$), and *serve*, while the sole accessor is *next*. The axioms are in Figure 4. Informally we'll write $\langle c_1, c_2 \ldots c_n \rangle$ for $\langle\rangle \lhd c_1 \lhd c_2 \ldots \lhd c_n$.

name	abbrev.	type
empty	$\langle\rangle$	$\mathbb{Q}A$
enqueue	$\_ \lhd \_$	$\mathbb{Q}A \to A \to \mathbb{Q}A$
append	$\_ +\!\!+ \_$	$\mathbb{Q}A \to \mathbb{Q}A \to \mathbb{Q}A$
serve		$\mathbb{Q}A \to \mathbb{Q}A$
next		$\mathbb{Q}A \to \text{Option } A$

Fig. 3. Queue Operations

$p +\!\!+ \langle\rangle$	$= p$
$p +\!\!+ (q \lhd a)$	$= (p +\!\!+ q) \lhd a$
serve $\langle\rangle$	$= \langle\rangle$
serve $(\langle\rangle \lhd a)$	$= \langle\rangle$
serve $(p \lhd a \lhd b)$	$=$ (serve $(p \lhd a)) \lhd b$
next $\langle\rangle$	$=$ None
next $(\langle\rangle \lhd a)$	$=$ Some a
next $(p \lhd a \lhd b)$	$=$ next $(p \lhd a)$

Fig. 4. Queue Axioms

There are well known techniques for showing sufficient completeness (e.g. [TM92][Chapter 12]). For queues we can build a *canonical term algebra* generated by the operators $\langle\rangle$ and $\lhd$, then group the axioms as indicated in Figure 4 to define total functions on the canonical term algebra. A straightforward structural induction suffices to show that $+\!\!+$ and *serve* map canonical queues (i.e. ones generated by $\langle\rangle$ and $\lhd$ and values of pre-existing types) to canonical queues, and that *next* maps canonical queues to ground terms.

Monads. We propose to apply sufficient completeness to monads as well.

Haskell's `IO` monad was the 'killer application' which introduced monads to programmers. One key feature of `IO` is that there are no accessors, hence the containment of effects, but also the admission that one does not know how `IO` behaves.

Many other monads, however, do have accessors. Figures 2 shows a sufficiently complete axiomatization of a state monad with a single accessor, *run*. It may seem unintuitive, but *get* is not an accessor, due to its type.

3 High-Level Proofs

This section sketches proofs for various OS features. Throughout this section, we assume an equivalence on computations. Later, §4.3 shows how to build this equivalence.

Hypothesis 3.1 *we have an equivalence relation, $\cong$, on computations. Moreover, it is a congruence with respect to bind ($\star$), and it is co-inductive: that is, an equivalence may safely be used in its own proof provided that each side is unrolled at least once.*

The high level separation proofs use co-induction. This technique shows *observational* properties of potentially infinite objects; that is, it shows that there is no finitary evidence contradicting the property. The idea is: if one unfolding of a goal reveals a similar subgoal but produces no evidence that the goal is false, then no amount of unfolding can produce such evidence.

§3.1 defines a simple scheduler and proves two simple separation results. §3.2 presents a more realistic scheduler with timer interrupts. §3.3 is more realistic in an orthogonal way, modelling page faults. Each subsection also discusses primitive operations and assumptions presumed to be part of the H-interface. The term *high level* distinguishes these proofs from mechanized ones in Coq. High level proofs convey meaning to a mathematician, even if details are sketchy, while mechanized proofs are thorough, to the point of being tedious.

3.1 Basic Round Robin

The round-robin scheduler, $\varrho : \mathbb{QC} \to \mathbb{M}1$, repeatedly cycles through the processes in the queue. It is used in two problems. Solo/Duo separation is the simplest problem, illustrating the primitives and the reasoning. The n-process round-robin with one POI shows how the complexity scales with problem size.

Hypothesis 3.2 *Assume an atomic step of execution $\epsilon : \mathbb{C} \to \mathbb{MC}$ which leaves process identifier associated with a context invariant.*

The computation ϵ plays two roles: first, it is the smallest computation managed by the scheduler, and second, it is the atomic step of machine execution.

Definition 3.3 (Local Respect). *A process $i : \mathbb{P}$ has local respect, denoted by* $\mathrm{LR}\, i$*, iff a step of its execution is unobservable by the equivalence. Specifically:*

$$\mathrm{LR}\, i := \forall c.i = \mathrm{ID}\, c \to \exists c'.\epsilon c \cong \eta c'$$

In addition, we overload the predicate LR *as follows:* $\mathrm{LR}\, c$*, for* $c : \mathbb{C}$*, means* $\mathrm{LR}\ (\mathrm{ID}\, c)$*, and* $\mathrm{LR}\, \vec{c}$*, for* $\vec{c} : \mathbb{QC}$ *means that for each* c *in* $\vec{c}$*,* $\mathrm{LR}\, c$*.*

Definition 3.4 (Round Robin Scheduler).

$$\varrho\ \vec{c} := \begin{cases} \eta\ () & \text{if next } \vec{c} = \mathrm{None} \\ \epsilon\ c \star \lambda c'.\varrho\ ((\mathrm{serve}\ \vec{c}) \lhd c') & \text{if next } \vec{c} = \mathrm{Some}\ c \end{cases}$$

Informally, for a nonempty queue this definition amounts to:

$$\varrho\ \langle c_1, c_2 \ldots c_n \rangle := \epsilon c_1 \star \lambda c_1'.\varrho\ \langle c_2 \ldots c_n, c_1' \rangle$$

or, more compactly: $\varrho\ \langle c, \vec{d} \rangle := \epsilon c \star \lambda c'.\varrho\ \langle \vec{d}, c' \rangle$.

Solo/Duo Separation. The solo and duo schedulers instantiate round-robin on queues of length 1 and 2 respectively.

Theorem 3.5 (Separation). $\forall c_1 c_2 : \mathbb{C}.\, \mathrm{LR}\, c_2 \to \varrho\ \langle c_1, c_2 \rangle \cong \varrho\ \langle c_1 \rangle$

Proof. Use co-induction. Fix c_1 and c_2, and assume $\mathrm{LR}\, c_2$. By $\mathrm{LR}\, c_2$ it follows that there is a c for which $\epsilon c_2 \cong \eta c$ and, moreover $\mathrm{LR}\, c$ holds because the step ϵ leaves the process ID unchanged. Now, comparing schedulings, observe that:

$$\begin{aligned}
&\varrho\ \langle c_1, c_2 \rangle \\
&\cong \epsilon c_1 \star \lambda c_1'.\epsilon c_2 \star \lambda c_2'.\varrho\ \langle c_1', c_2' \rangle && \text{(unfolding } \varrho \text{ twice)} \\
&\cong \epsilon c_1 \star \lambda c_1'.\eta c \star \lambda c_2'.\varrho\ \langle c_1', c_2' \rangle && \text{(local respect of } c_2\text{)} \\
&\cong \epsilon c_1 \star \lambda c_1'.\varrho\ \langle c_1', c \rangle && (\eta \text{ is the left identity of } \star) \\
&\cong \epsilon c_1 \star \lambda c_1'.\varrho\ \langle c_1' \rangle && \text{(coinductive hypothesis, guarded by } \epsilon c_1\text{)} \\
&\cong \varrho\ \langle c_1 \rangle && \text{(folding } \varrho\text{)}
\end{aligned}$$

***n*-Process Round Robin.** We next extend separation to a general round robin scheduler with one POI and any number of processes showing local respect. But first we need some lemmas.

Lemma 3.6 (Single Rotation). *For any context* c*, if* $\mathrm{LR}\, c$ *then there exists* c' *such that* $\mathrm{LR}\, c'$ *and for any queue of contexts,* $\vec{d}$*, we have* $\varrho\ \langle c, \vec{d} \rangle \cong \varrho\ \langle \vec{d}, c' \rangle$

Proof. Unfold the ϱ on the left, apply LR, and note η is the left identity of $\star$.

Lemma 3.7 (Multi-Rotation). *For any lists of context,* $\vec{d}$ *and* $\vec{e}$ *if* $\mathrm{LR}\, \vec{d}$ *then there is a* $\vec{d'}$ *for which* $\mathrm{LR}\, \vec{d'}$ *and* $\varrho\ \langle \vec{d}, \vec{e} \rangle \cong \varrho\ \langle \vec{e}, \vec{d'} \rangle$

Proof. By induction on the length of $\vec{d}$, using the Single Rotation Lemma.

Theorem 3.8 (Separation — Round Robin). *For any lists of contexts* $\vec{e}$*,* $\vec{d}$ *for which* $\mathrm{LR}\, \vec{e}$ *and* $\mathrm{LR}\, \vec{d}$*, and for any* c *we have* $\varrho\ \langle \vec{e}, c, \vec{d} \rangle \cong \varrho\ \langle c \rangle$

Proof. By Multi-rotation, unfolding and coinduction.

3.2 Timer Interrupts

This section sketches separation for a scheduler that executes without switching contexts until the hardware timer interrupts it.

As this is a *re*development of §3.1 with different details, none of Hypothesis 3.2 through Theorem 3.8 are available. Hypothesis 3.9 through Remark 3.19 are independent of that work.

Primitives. Recall that in §3.1 ϵ was both the smallest computation managed by the scheduler, and the atomic step of machine execution. For timer interrupts, we distinguish these two roles, keeping ϵ to denote the former, but for the latter pushing the interface downwards to reveal ϵ_1. Now take ϵ_1 to be atomic and build ϵ from it. Local respect is defined for ϵ_1 and then proven to extend to ϵ.

Hypothesis 3.9 *Assume an atomic step of execution* $\epsilon_1 : \mathbb{C} \to \mathbb{MC}$ *which leaves process IDs invariant.*

Definition 3.10 (Time Slice). $\epsilon : \mathbb{N} \to \mathbb{C} \to \mathbb{MC}$

$$\epsilon\ 0\ c := \epsilon_1 c \qquad\qquad \epsilon\ (S\ m)\ c := \epsilon_1 c \star \lambda c'.\epsilon\ m\ c'$$

Note that ϵ takes at least one step regardless of the value of n.

Definition 3.11 (Local Respect). $\mathrm{LR}\, i := \forall c.i = \mathrm{ID}\, c \to \exists c'.\epsilon_1 c \cong \eta c'$

If the equivalence is oblivious to one step of computation of a process, then it is oblivious to any finite sequence of steps.

Lemma 3.12 (Local Respect of ϵ). $\mathrm{LR}\, c \to \forall n.\exists c'.\epsilon n c \cong \eta c'$

Proof. Induction on n

Definition 3.13 (Oracles). *An oracle is an infinite stream of natural numbers. The type of oracles is denoted by* $\mathbb{N}^\omega$*. That is:* $\mathbb{N}^\omega := \nu X.\mathbb{N} \times X$

An *oracle*, as an additional argument to ϱ, models time-slices. Each number in the oracle determines how many ϵ_1-steps a process takes between context switches. This model is faithful, since for any real execution in hardware, there is a corresponding stream of naturals.

Definition 3.14 (Round Robin with Oracle).

$$\varrho : \mathbb{N}^\omega \to \mathbb{QC} \to \mathbb{M}1$$
$$\varrho\ (h,t)\ \langle c, \vec{d}\rangle := \epsilon\ h\ c \star \lambda c'.\varrho\ t\ \langle \vec{d}, c'\rangle$$

Separation needs a few lemmas for manipulating oracles. The proofs are omitted as they similar enough to the proof of Theorem 3.5. The first shows a correspondence between solo and duo schedulings: if $\mathrm{LR}\, d$ then any duo execution involving d, corresponds to a solo execution without d but using a different oracle. Specifically:

Lemma 3.15 (Correspondence). $\mathrm{LR}\, d \to \forall o.\varrho\ o\ \langle c, d\rangle \cong \varrho\ (\mathrm{alt}\ \ o)\ \langle c\rangle$ *where* alt *selects alternate elements of an oracle:* $\mathrm{alt}\ (h_1, (h_2, t)) := (h_1, \mathrm{alt}\ \ t)$

The next shows an independence of the behavior of a solo scheduling from the oracle it uses, but it is easier to establish as a corollary of a more general result.

Lemma 3.16 (Independence, generalized). $\forall n_1\ n_2\ o_1\ o_2.$

$$\epsilon\ n_1\ c \star \lambda c'.\varrho\ o_1\ \langle c'\rangle \cong \epsilon\ n_2\ c \star \lambda c'.\varrho\ o_2\ \langle c'\rangle$$

Corollary 3.17 (Independence). $\forall o_1\ o_2.\varrho\ o_1\ \langle c\rangle \cong \varrho\ o_2\ \langle c\rangle$

Theorem 3.18 (Separation). $\forall o_1\ o_2.\, \mathrm{LR}\, d \to \varrho\ o_1\ \langle c, d\rangle \cong \varrho\ o_2\ \langle c\rangle$

Proof. Both sides are equivalent to $\varrho\ (\mathrm{alt}\, o_1)\ \langle c\rangle$, by Independence and Correspondence, respectively.

Remark 3.19. It is possible to recast this work with an implicit oracle, incorporated into the monad. The landmark results (the definition of round-robin, the statement of separation) are then identical to those in §3.1, but the proof details differ.

3.3 Page Faults

Physical memory has two problems: it is limited, and it must be shared among processes. Yet most operating systems give user processes an abstraction of memory, called *virtual memory*, which emulates an unbounded amount of contiguous memory. It does so with a technique called *paging*. Memory is divided into uniformly sized *pages*. When all physical memory is in use, yet more is needed, one page of physical memory is moved to disk. When a page that resides on disk is needed in memory, space is found in physical memory.

In the following, the key feature is that a step may either advance as normal, or raise a *page fault* which halts normal execution, invokes a kernel process to page the needed data from disk to memory, and then resumes execution.

The separation argument extends to schedulers with a restricted form of page fault. Page maps, drawn from an abstract type $\mathbb{V}$, are explicitly present as an extra argument/return value to atomic step, ϵ, and the scheduler, ϱ. An abstract type of memory addresses, $\mathbb{A}$, is also explicit.

Primitives. As in §3.2, we push the interface downwards to work with lower level primitives. In this case there are two primitive computations. The first is an atomic step of execution, ϵ_f, that may either succeed, returning an updated context, or raise a page fault, returning the address that is not currently mapped. The second, ξ, extends a page map to include a specific address.

Hypothesis 3.20

$$\epsilon_f : \mathbb{V} \to \mathbb{C} \to \mathbb{M}(\mathbb{C} + \mathbb{A})$$
$$\xi : \mathbb{V} \to \mathbb{A} \to \mathbb{M}\mathbb{V}$$

The memory system for virtual memory is responsible for paging between memory and disk. For purposes of separation, ξ is a primitive, given by the memory system. We require an assumption, similar to local respect, that the behavior of ξ is not observable:

Hypothesis 3.21 $\forall v.\exists v'.\xi v a \cong \eta v'$

Handling Single-Faultedness. The first step is to define a non-faulting step of execution, again called ϵ, for which we can prove separation in the style of §3.1. A *handled step* of execution ϵ_h combines ϵ_f with a page fault handler of type $\mathbb{V} \to \mathbb{C} \to \mathbb{A} \to \mathbb{M}(\mathbb{V} \times \mathbb{C})$

$$\epsilon_h : \mathbb{V} \to \mathbb{C} \to (\mathbb{V} \to \mathbb{C} \to \mathbb{A} \to \mathbb{M}(\mathbb{V} \times \mathbb{C})) \to \mathbb{M}(\mathbb{V} \times \mathbb{C})$$

$$\epsilon_h\ v\ c\ h := \epsilon_f\ v\ c \star \lambda x. \begin{cases} \eta\ (v, c') & \text{if } x = \iota_1 c' \\ h\ v\ c\ a & \text{if } x = \iota_2 a \end{cases}$$

The result is robust to page faults, though we don't yet have a handler h. The details of h depend upon the nature of page faults. For concreteness, we make the following assumption.

Hypothesis 3.22 (single-faultedness) *If one step of the execution of context c with page map v faults, specifying address i, and if extension of v via ξ to include i yields page map v', then retrying that step on v' will not fault.*

Given single-faultedness, a robust step function can be coded using ϵ_h, and letting $\perp$ stand for an unreachable computation:

$$\epsilon : \mathbb{V} \to \mathbb{C} \to \mathbb{M}(\mathbb{V} \times \mathbb{C})$$

$$\epsilon\ v\ c := \epsilon_h\ v\ c\ (\lambda v.\lambda c.\lambda i.\xi\ v\ i \star \lambda v'.\epsilon_h\ v'\ c\ \perp)$$

Note that ϵ_h occurs twice here, with the second occurrence being the handler for the first. The single-faultedness of ξ guarantees that the second use is robust despite the occurrence of $\perp$. The sketch of separation is: from local respect for ϵ_f and ξ, show local respect for ϵ_h and for ϵ; the proof of separation for Theorem 3.5 then goes through with minor modifications.

More General Faults. The single-faultedness assumption is simplistic, but the approach scales. If the only memory access is through load and store then there are at most two faults: one accessing the instruction, and one accessing the data. If a machine operation like `ADD` can access indirect memory then there may be numerous faults due to both the number of arguments and the indirect lookup. If the number of faults is bounded, then a generalization to n-faultedness suffices. Alternatively, using the iteration monad of §4.4, ϵ_f can loop until it succeeds.

Remark 3.23. Page faults are not yet integrated with timer interrupts. To do so requires $\epsilon_f : \mathbb{V} \to \mathbb{C} \to \mathbb{M}(\mathbb{C} \times \mathit{Option}\ \mathbb{A})$, which may do non-trivial work before the interrupt, so it always returns an updated context.

3.4 Summary of High-level Proofs

The proofs for full round-robin, timer interrupts and page faults are independent generalizations of the Solo/Duo scheduler problem. It remains to combine all three generalizations into the same system. We see no fundamental impediment to combining them, except for an increase in complexity. We anticipate that having mechanized theorem prover support will be beneficial for that integration. Next we turn to issues in mechanization.

4 Mechanization Design

The eventual goal of this research is to mechanically verify separation for Haskell code. Mechanization is necessary to verify actual code, and to deal with tedious details that arise as schedulers become more complex.

To compensate for the lack of logic in Haskell, we embed the scheduler into Coq. To support the custom equivalence relation $\cong$, which is a central feature of separation proofs, we use setoids with an applicative structure instead of mere types. The mechanization strategy leverages the monadic nature of the scheduler in two different ways. First, as an ADT for computations. Second, to add effects: logging, which is the basis for $\cong$, and non-termination, which is a work-around for a difficulty in Coq.

4.1 Embedding Haskell in Coq

By an *applicative structure* we mean an implementation of typed λ-calculus. Terms may be built by function application and λ-abstraction. Terms evaluate by β-reduction. Each term has a type, and there is an *arrow* type constructor, so that $A \to B$ classifies functions from type A to type B.

The features of Haskell that the scheduler uses are the statically typed applicative structure, non-terminating tail recursion, and the monadic interface to hardware. Coq implements a simply typed λ-calculus augmented by inductive and co-inductive types. It provides in an integrated framework both an executable language of types & terms, and a language of properties & proofs. To maintain logical consistency, recursion is restricted in Coq, but it is, in principle, still powerful enough to implement a scheduler.

At first glance, Coq is the ideal target for the embedding: inductive types can model the hardware monad, tail recursion maps to co-recursion, and the applicative structure maps directly. It turns out that we both need a novel applicative structure, as types are too poor to model our equivalence, and we need a novel means of iteration, as co-recursion is difficult to use.

4.2 Setoids

Now we discuss the change to the applicative structure. The notion of equality in Coq is Leibniz equality: two terms are equal iff no context can distinguish

them. Separation requires a coarser equivalence. Leibniz equality distinguishes ϵd from $\eta d'$, as ϵd has effects, while $\eta d'$ does not: when we assert $\mathrm{LR}\, d$, we need to equate these two computations.

To model an equivalence relation on a type, Coq uses a *setoid*, which is a type paired with an equivalence relation. An operator on a setoid must be a *congruence*, that is a function respecting the equivalence: if $f : A \to B$ and $a_1 \equiv_A a_2$ then $f a_1 \equiv_B f a_2$. Each definition of a λ-term requires a proof of congruence. We'll use the term A-*equivalence* to refer to $\equiv_A$.

In this setting, $\mathbb{M}$ is a function at the type level which maps any setoid to another setoid. That is, given a type A, an equivalence $\equiv_A$ as well as proofs of reflexivity, symmetry, transitivity of $\equiv_A$, a monad $\mathbb{M}$ will generate at least five new items: a type of computations (denoted by $\mathbb{M}A$), an equivalence ($\equiv_{\mathbb{M}A}$, which is based on $\equiv_A$), and proofs of each of reflexivity, symmetry and transitivity for $\equiv_{\mathbb{M}A}$. $\mathbb{M}$ may introduce other operations (e.g. *put* and *get*) and establish more laws. Each operation (e.g. η, $\star$, *put*, and *get*) must be a congruence.

4.3 Logging Monad

Logging is key to implementing the coarse-grained equivalence of separation proofs. For our purposes, the interface is not important; what matters is that it provides a way to instrument *other* monads without changing *their* interfaces.

A logging computation is parameterized by a type of *token*. Intuitively, an A-computation is a finite list of these tokens, ending in a value from A. Return builds an empty list of tokens plus a pure A-value. Bind concatenates two lists, where the second list depends on the A-value embedded in the first. A new operation, $log : T \to \mathrm{Log}\ 1$, creates a singleton list from a token.

Formally, fix parameter T, then the fixed point $\mathrm{Log}\, A := \mu X.A + T \times X$ defines the type of computation. Using $Val\ a := \iota_1\ a$ and $Step\ t\ m := \iota_2\ (t, m)$ as mnemonic constructors:

$$\begin{aligned} \log t &:= \mathrm{Step}\, t\ (\mathrm{Val}\ ()) & \mathrm{Val}\, a \star r &:= r\ a \\ \eta\ a &:= \mathrm{Val}\, a & (\mathrm{Step}\, t\ q) \star r &:= \mathrm{Step}\, t\ (q \star r) \end{aligned}$$

It is straightforward to show that the monad laws hold. Two computations are $\mathrm{Log}\, A$-equivalent iff their return values are A-equivalent and their token lists are equivalent (that is, the lists have the same length and T-equivalent entries occur in corresponding positions).

Instrumentation. The value of logging is in combination with other monads. Given a set of process identifiers, assert for which ones local respect holds. Given a monad $\mathbb{U}$ with operation $\epsilon : \mathbb{C} \to \mathbb{U}\mathbb{C}$ create an instrumented monad $\mathbb{M}A = \mathbb{U}\ (LogA)$ by instantiating T to $\mathbb{C}$, then instrumenting a step of execution:

$$\epsilon_{\mathbb{M}}\ c := \begin{cases} \epsilon\ c & \text{if } \mathrm{LR}\, c \\ \log\ c \gg \epsilon\ c & \text{if } \neg(\mathrm{LR}\, c) \end{cases}$$

The coarsened equivalence relation on $\mathbb{M}$ is then the relation of 3.1.

Distributivity. The logging monad is distributive: its effects can be combined with those of any other monad $\mathbb{U}$ because it has an operator, $\delta : \forall A.\, \mathrm{Log}\ (A) \to \mathbb{U}(\mathrm{Log}\ A)$, satisfying laws of interaction with the *join*, *map* and η operations of the two monads.

$$\delta\ (\mathrm{Val}\, u) := \mathrm{map}_{\mathbb{U}}\ \mathrm{Val}\ \ u \qquad\qquad \delta\ (\mathrm{Step}\, t\ m') := \mathrm{map}_{\mathbb{U}}\ (\mathrm{Step}\, t)\ m'$$

4.4 Iteration Monad Transformer

One limitation of Coq is its restriction on recursion. To relax the restriction for monadic code, we add the ability to loop as a monadic effect, which is sufficient for OS code. The additional interface is a monad transformer which adds an iteration operator [BÉ93]. That is, it is a monad $(\mathbb{M}, \eta_{\mathbb{M}}, \star_{\mathbb{M}})$ parameterized by an underlying monad $(\mathbb{U}, \eta_{\mathbb{U}}, \star_{\mathbb{U}})$, where $\mathbb{M}$ gains the effects of $\mathbb{U}$ through lifting, while $\mathbb{M}$ also adds the effect of non-termination by adding a *loop* operation.

name	abbrev.	type
lift	$\_^{\dagger}$	$\mathbb{U}A \to \mathbb{M}A$
loop		$(A \to \mathbb{M}(A + R)) \to A \to \mathbb{M}R$
unroll		$\mathbb{M}A \to \mathbb{N} \to \mathbb{U}(\mathrm{Option}(\mathbb{N} \times A))$

Fig. 5. Iteration Operations

$$\eta_{\mathbb{M}}\ a = \mathrm{lift}\ (\eta_{\mathbb{U}}\ (\mathrm{Some}\ \ a))$$

$$\mathrm{unroll}\ (\mathrm{lift}\ \ u)\ n = u \star_{\mathbb{U}} \lambda a.\eta_{\mathbb{U}}\ (\mathrm{Some}\ (n, a))$$

$$\mathrm{unroll}\ (p \star_{\mathbb{M}} q)\ n = \mathrm{unroll}\ \ p\ n \star_{\mathbb{U}} \lambda x. \begin{cases} \eta_{\mathbb{U}}\ \mathrm{None} & \text{if } x = \mathrm{None} \\ \mathrm{unroll}\ (q\ a)\ m & \text{if } x = \mathrm{Some}\ (m, a) \end{cases}$$

$$\mathrm{unroll}\ (e^{\dagger} a)\ n = \begin{cases} \eta_{\mathbb{U}}\ \mathrm{None} & \text{if } n = 0 \\ \mathrm{unroll}\ (ea \star_{\mathbb{M}} [e^{\dagger}, \eta_{\mathbb{U}}])\ m & \text{if } n = S\ m \end{cases}$$

Fig. 6. Iteration Axioms

The accessor *unroll* provides a basis for defining $\equiv_{\mathbb{M}A}$ in terms of $\equiv_{\mathbb{U}A}$. First, define a preorder on $\mathbb{U}$ computations: for $u_i : \mathbb{U}A_i$ let $u_1 \sqsubseteq_{\mathbb{U}} u_2$ mean $\exists q. u_1 \star q \equiv_{\mathbb{U}A_2} u_2$. Next, lift it to a preorder on $\mathbb{M}$: for $m_i : \mathbb{M}A_i$ let $m_1 \sqsubseteq_{\mathbb{M}} m_2$ mean $\forall n_1. \exists n_2.\, \mathrm{unroll}\ \ m_1\ \ n_1 \sqsubseteq_{\mathbb{U}} \mathrm{unroll}\ \ m_2\ \ n_2$. To define equality, given $m_i : \mathbb{M}A, i \in \{1, 2\}$ let $m_1 \equiv_{\mathbb{M}A} m_2$ mean $m_1 \sqsubseteq_{\mathbb{M}} m_2$ and $m_2 \sqsubseteq_{\mathbb{M}} m_1$ and the return values, if any, of m_1 and m_2 are equal.

We have the following results (proofs omitted due to space constraints).

Theorem 4.1. *if* $(\mathbb{U}, \eta_{\mathbb{U}}, \star_{\mathbb{U}})$ *is a monad then* $(\mathbb{M}, \eta_{\mathbb{M}}, \star_{\mathbb{M}})$ *is a monad*

Theorem 4.2. *The axiomatization of iteration is sufficiently complete.*

Remark 4.3. Arguably, hardware informally is a model as each unrolling corresponds to a backwards local `jmp` in hardware.

It is possible to implement a model of the iteration monad in Coq by taking η, $\star$, $lift$, and *loop* as constructors, and to define *unroll* recursively, though showing that *unroll* is well-founded is a little challenging.

Guardedness. Iteration theory is an alternative to using co-induction in Coq, which has proven difficult to work with. Coq uses a very conservative syntactic test to ensure that co-recursion is sound, however many simple functions violate this constraint, as well as proofs that mix induction with co-induction, as Multi-Rotation does. In general, using co-induction in Coq requires a deal of insight and creative refactoring of problems. There is active research into better support for coinductive proofs in Coq, and better programming styles to interact with co-induction (for instance [BK08], [DA09]) but it is not yet at the state we require.

Co-inductive proofs are more readable than ones using approximation lemmas or pre-orders, but at the time of writing, it seems preferable to mechanize using the latter. Our earlier attempts at mechanization depended on co-induction in Coq, and this kept us from progressing.

5 Conclusions

5.1 Summary & Future Work

This paper formulates a way of framing the separation problem, shows a variety of high level, but simple separation proofs (n-process round robin, timer interrupts, page faults), and sketches a way of mechanizing those proofs, and others, in Coq. Mechanization is essential for two reasons: first, the ultimate goal of this research is to verify code, and second the amount of proof detail increases with the scheduler complexity and with H-monad complexity. The hypotheses in this work are drastic simplifications of the actual H-monad.

We also discussed a strategy for mechanization, comprising an embedding of Haskell code into Coq, albeit with an applicative structure on setoids instead of using native types. We use monads to define an appropriate equivalence relation, $\cong$, and to work around limits with Coq's implementation of coinduction. Finally, we use sufficient completeness to justify why a proof about a model can demonstrate a property about real hardware.

This is a work in progress, and much work remains. Foremost, the mechanization is only partially done. Second, we must use more realistic schedulers and interfaces; a start is to combine n-process round robin, timer interrupts and page faults into one separation problem. We also need a sufficiently complete axiomatization for a monad of timer interruptions. Some technical work remains:

We conjecture that the five laws of iteration theory hold (see [BÉ93]) though we have not yet finished all five proofs.

One reviewer suggested looking at *effect terms* and imposing equational constraints on the resulting Lawvere theories [PP08]. Logging, as presented here, is an ad hoc mechanism that achieves some of what effect terms do. Preliminary tests with effect terms show they are a more direct and versatile way of recording the actions of a computation, so we hope to integrate them in future work. For instance it may be fruitful to treat non-determinism as a binary (or $\mathbb{N}$-ary) term constructor instead of an oracle. Unfortunately, effect terms still violate Coq's guardedness constraint

5.2 Related Work

Rushby [Rus92] provides an automata-theoretic means of showing noninterference. His basic vocabulary includes three notions: local respect, step consistency, and output consistency. We keep all three, explicitly adopting a variant of the first, while the two notions of consistency are implicitly maintained by setoids. In general, his technique is quite powerful, addressing intransitive non-interference. We address neither transitivity nor intransitivity. On the other hand, his work only applies to models. Our work is based on the idea of a monad, and lends itself to proofs about actual code.

References

[BÉ93] Stephen L. Bloom and Zoltán Ésik. *Iteration Theory: The Equational Logic Of Iterative Processes.* EATCS monographs on theoretical computer science. Springer-Verlag, 1993.

[BK08] Yves Bertot and Ekaterina Komendantskaya. Using structural recursion for corecursion. In Stefano Berardi, Feruccio Damiani, and Ugo de Liguoro, editors, *Types 2008*, volume 5497 of *Lecture Notes in Computer Science*, pages 220–236. Springer-Verlag, 2008.

[DA09] Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction, 2009. Draft.

[HJLT05] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 116–128, 2005.

[Mac71] Saunders Mac Lane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics. Springer-Verlag, 1971.

[PP08] Gordon Plotkin and Matija Pretnar. A logic for algebraic effects. In *LICS '08: Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 118–129, 2008.

[Rus92] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, Stanford Research Institute, December 1992.

[TM92] J. L. Turner and T. L. McCluskey. *The Construction of Formal Specifications: An Introduction to the Model-Based and Algebraic Approaches.* McGraw Hill, 1992. Online at `http://scom.hud.ac.uk/scomtlm/book.pdf`.

Software Institutions

Adis Hodzic[1] and Magne Haveraaen[2]

[1] Faculty of Engineering, Bergen University College, PB 7030, 5020 Bergen, Norway
Adis.Hodzic@hib.no
[2] Department of Informatics, University of Bergen, PB 7800, N-5020 Bergen, Norway
http://www.ii.uib.no/~magne/

Abstract. The purpose of institutions [GB92] is to formalise the notion of model theory for logics, relating signatures (interfaces), formulas (axioms), and models. The use of algebraic specifications for defining software falls in this framework, with set-theoretic models being the norm. This gives only an indirect relationship between specifications and software. Here we investigate a way of using software as models in an institution. This gives some insight in the expressive power of classical programming concepts.

1 Introduction

As the size of industrial scale software is increasing, the need for modularisation of software is seen as more and more acute. This has led to the recent development of many new structuring mechanisms: templates (generic classes) [LSAS77,Wag91,Str97,BOSW98], generative programming [CE00], aspect oriented programming [KLM+97], coordination [Fia04], etc. Many of these mechanisms are supported by program transformation tools, and in some cases this manipulation, like macro expansion, is seen as a purely syntactic operation. In order to better understand such methods, we need the ability to see software as a formal mathematical entity which can be manipulated by well understood techniques. Thus we need to approach software as a two-level structure — both on the syntactic and on the semantic level. When modularisation and software building operations on these two layers are coordinated, we will have a well-behaved technology.

Using algebraic specification technology the relationship between syntax and semantics is handled in a precise way. In algebraic software development techniques the relationship between programs and specifications is typically interpreted in one of two ways:

- staying at an algebraic level, seeing a program as the endpoint of a refinement sequence, where a polymorphic specification gradually is turned into a monomorphic specification ([ST97]), or
- programs and algebras are independent entities, and their connection is an indirect relationship where the semantics of the program is verified against the class of models defined by the specification ([GH93]).

In the algebraic literature the former of these seems to dominate.

Here we will approach programs and implementations of programs directly as models of specifications. We use a very simplified notion of programs (=algorithms+data structures [Wir76]) - as terms and collections of attribute values, but where the base types are given as an arbitrary signature. We will then show that we may build an institution with such programs as models. Notice that, on such models we may define semantical notions using normal algebraic techniques, but, in addition, due to the syntactical nature of models, other program properties (such as run-time cost of a program) may be defined and examined.

Our approach seems closely related to that of Fiadeiro [Fia04], which has a library based approach (using signatures) to defining components. These are then combined using categorical constructs for building software. Wagner [Wag90] and Walters [Wal91] use distributive categories to define semantical notions of programs and program executions and relate these to specifications.

This paper is organised as follows. In section three we define the notion of programs in a programming language with a support library, showing how to build an institution with programs as models. Finally we conclude, listing some future work based on this framework.

2 Preliminary Concepts

Let **Set** be the category of sets and total functions, **CAT** the category of all categories, and **Alg** the category of total many-sorted algebras and total homomorphisms.

Definition 1 (Institution). *An* institution *is given by a quadruple* $\mathcal{INST} = (\mathbf{Sign}, \mathit{Sen}, \mathit{Mod}, \models)$*, where*

- **Sign** *is a category of* signatures*,*
- $\mathit{Sen} : \mathbf{Sign} \to \mathbf{Set}$ *is a functor giving the* sentences
- $\mathit{Mod} : \mathbf{Sign}^{op} \to \mathbf{CAT}$ *is a functor giving the category of* models
- $\models$ *is a family of* satisfaction relations*:* $\models_\Theta \subseteq \mathit{Mod}(\Theta) \times \mathit{Sen}(\Theta)$ *for each* $\Theta \in |\mathbf{Sign}|$

such that, for each morphism $\theta \in \mathbf{Sign}(\Theta, \Theta')$*, the* satisfaction condition*,*

$$M' \models_{\Theta'} \mathit{Sen}(\theta)(\varphi) \Leftrightarrow \mathit{Mod}(\theta)(M') \models_\Theta \varphi,$$

holds for each $M' \in |\mathit{Mod}(\Theta')|$ *and each* $\varphi \in \mathit{Sen}(\Theta)$*.*

A standard example is the institution of total many-sorted equational specifications. A signature $\Theta = (S, F)$ consists of a set of sorts S and function declarations F with argument lists being strings from S and result type a sort from S. Signature morphisms $\theta : \Theta \to \Theta'$ map sorts to sorts, function declarations to function declarations, preserving the sorting of arguments and result types. We have sets of variables sorted by S, and as terms we use all type-correct expressions formed from variables V and functions declared in F.

Definition 2 (Total many-sorted equational specifications). *The institution $\mathcal{TEL}$ consists of:*

- *Signature category: the category of total many-sorted signatures* **Sig** *and signature morphisms. For any signature $\Theta = (S, F)$ and variables V sorted by S there is the standard notion of sorted terms $t \in T(\Theta, V)$.*
- *Sentences: $Sen(\Theta)$ for a signature Θ are equations, i.e., triples $\langle V, t_1, t_2 \rangle$ of variables V and terms $t_1, t_2 \in T(\Theta, V)_s$ for some $s \in S$. Sentence translation $Sen(\theta : \Theta \rightarrow \Theta')$ is systematic renaming of the operations in $t \in T(\Theta, V)$ according to θ.*
- *The models $A : \Theta \rightarrow$* **Set** *for a signature Θ are given by*
  - *Objects: (algebras) $A \in |\mathbf{Alg}(\Theta)|$, defining a set $A(s)$ for every sort $s \in S$, a product set $\Pi_{r \in R} A(r)$ for every family R of S, and a total function $A(f) : \Pi_{r \in \mathrm{dom}(f)} A(r) \rightarrow A(\mathrm{cod}(f))$ for every $f \in F$ with a family of argument sorts $\mathrm{dom}(f) \subseteq S$ and result sort $\mathrm{cod}(f) \in S$.*
  - *Morphisms: (homomorphisms) $h \in \mathbf{Alg}(\Theta)(A, B)$, i.e., an S-indexed collection of total functions $h_s : A(s) \rightarrow B(s)$ for $s \in S$ such that given any $f \in F$, for all $\Pi_{r \in \mathrm{dom}(f)} a_r \in A(\mathrm{dom}(F))$ the homomorphism property is satisfied: $h(\mathrm{cod}(f))(A(f)(\Pi_{r \in \mathrm{dom}(f)} a_r) = B(f)(\Pi_{r \in \mathrm{dom}(f)} h(r)(a_r))$.*

  For every signature morphism $\theta : \Theta \rightarrow \Theta'$ where $\Theta = (S, F)$ we have the retract $\mathbf{Alg}(\theta) : \mathbf{Alg}(\Theta') \rightarrow \mathbf{Alg}(\Theta)$, such that for an algebra A' for Θ', the algebra $A = \mathbf{Alg}(\theta)(A')$ is defined by $A(s) = A'(\theta(s))$, $s \in S$, and $A(f) = A'(\theta(f))$, $f \in F$, and a homomorphism $h' \in \mathbf{Alg}(\Theta')(A, B)$ defines a homomorphism $h \in \mathbf{Alg}(\Theta)(\mathbf{Alg}(\theta)(A'), (\mathbf{Alg}(\theta)(B'))$ by $h_s = h'_{\theta(s)}$, $s \in S$.
- *The relation $A \models_\Theta \langle V, t_1, t_2 \rangle$ is satisfied when for all assignments $\alpha : V \rightarrow A$ we have that $A(\alpha)(t_1) = A(\alpha)(t_2)$, where $A(\alpha)(t)$ is the evaluation of the term $t \in T(\Theta, V)$ in the algebra A with the assignment α.*

Definition 3 (Variable set). *Given a signature $\Theta = (S, F)$, a variable set is a pair $(V, sort_V : V \rightarrow S)$.*

Further in the text, we will use V to denote variable set $(V, sort_V)$. For variables V over a signature $\Theta = (S, F)$ we write $\mathrm{sort}_V(x) = s$ or $x : s$ to indicate that the variable $x \in V$ is of sort $s \in S$. The function $tsort_V : T(\Theta, V) \rightarrow S$ will denote expansion of a $sort_V$ to terms, $tvar_V : T(\Theta, V) \rightarrow Pow(V)$ will denote the function that takes a term t and returns a set of variables used in t. When no confusion occurs, we may drop index V. Given variables V and V' over Θ, then a substitution $\sigma : V' \rightarrow T(\Theta, V)$ extends to a substitution $\sigma : T(\Theta, V') \rightarrow T(\Theta, V)$. Composition of extended substitutions is associative.

For a signature $\Theta = (S, F)$, variables V over Θ, and algebra A, a term $t \in T(\Theta, V)_s$, $s \in S$, defines a function $A(t) : \Pi_{v \in V} A(\mathrm{sort}(v)) \rightarrow A(s)$ given by $A(t)(a) = A(\alpha_a)(t)$, $a = \Pi_{v \in V} a_v \in \Pi_{v \in V} A(\mathrm{sort}(v))$, where $\alpha_a : V \rightarrow A$ is $\alpha_a(v) = a_v$, $v \in V$.

Hereafter, we will restrict ourselves to the $\mathcal{TEL}$ signature notion **Sig** and related $\mathcal{TEL}$ sentences *Sen*. Thus when we write signature, signature morphism or sentence, this refers to the concepts of $\mathcal{TEL}$ rather than the general notions from the institution concept.

3 Program Institutions

Current software development is centred around the use of library packages. In a language like Java [GJS96] there seems to be a package available for most common tasks. Writing a program based on the appropriate combination of libraries, significantly reduces the amount of program text needed to be written by the programmer.

In our model of programs we take the perspective that a programming language consists of constructors for building data structures and constructors for building algorithms on top of a signature which captures the interface to the chosen library packages.

We assume that the interfaces of all the packages can be integrated to a standard many-sorted signature Σ. This signature also captures the built-in types and operations of the language itself, avoiding special treatment of these. A category of programs may then be defined from Σ and the programming language constructions. This category is syntax-oriented since programs are written as structured textual entities.

The semantics of the library signature Σ is given by an object (algebra) $A \in |\mathbf{Alg}(\Sigma)|$. We will not choose any specific algebra, as there may be, albeit small, semantical variation between various versions of a compiler. Likewise, there may be, at times large, semantical variations between various implementations of the library packages.

Writing a new package can be seen as defining an interface, the signature Θ, and providing an implementation for it, i.e., data structures for the sorts and algorithms for the operations. This can be defined using mappings to the category of programs.

In this section we will represent three programming categories that are considered in this paper: $\mathbf{Prog}_T(\Theta)$, $\mathbf{Prog}_{:=}(\Theta)$ and $\mathbf{Prog}_{:=/\sim}(\Theta)$. Other choices are possible, depending on the language we want to consider. $\mathbf{Prog}_T(\Theta)$ category contains some aspects of a programming language (variables, terms) while others are omitted (declaration and deletion of variables, order in which variables are substituted). Yet, the category is powerful enough to show how we can build the institution with programs as models. The other two categories resemble more closely to an imperative programming languages.

3.1 Term Program Category

The program category is based on a simplified programming language. Collections of *attributes*, defined as sorted sets of variables, are the only data structure constructions. Algorithms will be defined by well-typed expression terms. This is a very simple programming language, yet it is powerful enough to show the ideas of using program categories as models. Since it is built on top of an arbitrary signature, we can still have most of our expressive power given by the primitive sorts and operations on these.

Definition 4 (Term Program category $\mathbf{Prog}_T(\Theta)$). *The* program category $\mathbf{Prog}_T(\Theta)$ *for a signature* $\Theta = (S, F)$ *is given by*

- *Objects: a* data structure V *is a set of variables* V *over* Θ.
- *Morphisms: an* algorithm, $a : V_1 \to V_2$ *is a triple* $\langle V_1, V_2, \sigma : V_2 \to T(\Theta, V_1)\rangle$ *where* $\sigma : V_2 \to T(\Theta, V_1)$ *is a substitution.*
- *Composition: the composite algorithm* $a'' = (a; a') : V_1 \to V_3$ *for morphisms* $a = \langle V_1, V_2, \sigma : V_2 \to T(\Theta, V_1)\rangle : V_1 \to V_2$ *and* $a' = \langle V_2, V_3, \sigma' : V_3 \to T(\Theta, V_2)\rangle : V_2 \to V_3$ *is given by* $a'' = \langle V_1, V_3, \sigma'' : V_3 \to T(\Theta, V_1)\rangle$ *where* $\sigma'' = \sigma \circ \sigma'$.

This gives an associative composition where the identity morphism is given by identity substitution.

Thus an algorithm is identified by its inputs V_1, its outputs, V_2, and for each output variable the code (term) which computes it. Two data structures are isomorphic if there is a renaming of attribute variables between them.

Example 5. Given the signature $\Theta = (\{int\}, \{+ : int, int \to int\})$, the triple $\langle\{x, y\}, \{z\}, \sigma(z) = x * y + x * y\rangle$ is an example of a morphism between objects $\{x, y\}$ and $\{z\}$ in $\mathbf{Prog}_T(\Theta)$.

Categorical properties of $\mathbf{Prog}_T(\Theta)$. Model category of any signature in $\mathcal{TEL}$ is complete and cocomplete. This means that we may build new models out of the existing ones. Later in 3.6, will see that positive categorical properties of $\mathbf{Prog}_T(\Theta)$ will also be preserved by a model category for any signature in the institution that we are building (with $\mathbf{Prog}_T(\Theta)$ in its basis), so we want to investigate categorical properties of $\mathbf{Prog}_T(\Theta)$.

Proposition 6 (Program products). *Given a signature* Θ.

- *A product* $\Pi_{x\in X} V_x \in |\mathbf{Prog}_T(\Theta)|$, *over an index set* X *for data structures* $V_x \in |\mathbf{Prog}_T(\Theta)|$, $x \in X$, *is given by* $\uplus_{x\in X} V_x$, *the disjoint union of attributes* V_x, $x \in X$.
- *The projections are* $\langle \uplus_{x\in X} V_x, V_y, \sigma : V_y \to T(\Theta, \uplus_{x\in X} V_x)\rangle$ *for every* $y \in X$, *where* $\sigma(v) = v$ *is the injection of each* $v \in V_y$ *into the disjoint union.*
- *The mediating morphism for* $a_y = \langle V, V_y, \sigma_y : V_y \to T(\Theta, V)\rangle : V \to V_y$, $y \in X$, *is* $a = \langle V, \uplus_{x\in X} V_x, \sigma : \uplus_{x\in X} V_x \to T(\Theta, V)\rangle : V \to \uplus_{x\in X} V_x$ *where* $\sigma(v) = \sigma_y(v)$ *for* $v \in V_y \subseteq \uplus_{x\in X} V_x$.

Proof. Straight forward. □

Proposition 7 (Terminal objects). *The terminal object is the empty set of variables.*

Proof. The empty substitution is the unique morphism into the empty set of variables. □

Proposition 8 (Coequalizers). *Category* $\mathbf{Prog}_T(\Theta)$ *has all coequalizers.*

Proof. For two syntactically different substitutions $f, g : V' \to T(\Theta, V)$ there exist substitution $e : V'' \to T(\Theta, V')$ defined by $V'' = \{v \in V' | f(v) = g(v)\}$ and $e(v) = v$ for all $v \in V''$ is coequalizer. □

Proposition 9 (Program coproducts). *Category* $\mathbf{Prog}_T(\Theta)$ *has all (non-empty) coproducts.*

Proof. The coproduct for objects $A_y \in |\mathbf{Prog}_T(\Theta)|$, $y \in I$, is given by the object $CP_{A_y,y\in I} = \{v_w | w \in \Pi_{y\in I} T(\Theta, A_y)\}$. Arrow $i_{A_y} : A_y \to CP_{A_y,y\in I}$ is given by substitution $\sigma_{i_{A_y}}(v_{\overline{t}}) = t_y$ where $\overline{t}$ is an element of $\Pi_{y'\in I} T(\Theta, A_y)$ and has t_y as its y-th component. For any object $X \in |\mathbf{Prog}_T(\Theta)|$ and set of morphisms f_y, $y \in I$, mediating morphism $m_{X,f_y,y\in I} : CP_{A_y,y\in I} \to X$ is given by the substitution $\sigma_{m_{X,f_y,y\in I}}(v) = v_{\overline{t}}$ where for any $y \in I$, y-th component of $\overline{t}$ is term t_y such that underlying substitution of f_y has $\sigma_{f_y}(v) = t_y$.
Coproduct properties are satisfied:

- The compositionality requirement is satisfied: $m_{X,f_y,y\in I}(v) = \ldots \times f_y(v) \times \ldots$ and $i_{A_y}(\ldots \times f(v) \times \ldots) = f_y(v)$ for any $v \in X$ and any $y \in I$ so $i_{A_y}; m_{X,f_y,y\in I} = f_y$.
- The uniqueness of mediator requirement is satisfied: any different mediating morphism m' would mean that there exist at least one v such that $m'(v) \neq m_{X,f_y,y\in I}(v)$. But then at least one i_{A_y} for some $y \in I$ would have $f_y(v) \neq i_{A_y}; m'(v)$ so m' would not be mediator.

For two syntactically different substitutions $f, g : V' \to T(\Theta, V)$ there exist no substitution e such that $e; f$ and $e; g$ are syntactically equal, so category does not have all equalizers. Also, consider a signature with at one sort s and two constants c and c'. For the object $V = \{x\}$ with $sort(x) = s$, there exist at least two distinct substitutions (one with $x = c$ and the other with $x = c'$) from any other object V' of $|\mathbf{Prog}_T(\Theta)|$ into V. This means that initial objects do not exist.

Set-based semantics of $\mathbf{Prog}_T(\Theta)$.

Proposition 10. *An algebra* $A : \Theta \to \mathbf{Set}$ *for a signature* Θ *extends to a functor* $\overline{A} : \mathbf{Prog}_T(\Theta) \to \mathbf{Set}$ *given by*

- $\overline{A}(V) = \Pi_{v\in V} A(sort(v))$ *for every set of variables* V *over* Θ.
- $\overline{A}(a : V_1 \to V_2) = f : A(V_1) \to A(V_2)$ *for* $a = \langle V_1, V_2, \sigma : V_2 \to T(\Theta, V_1)\rangle$, $f = \langle A(\sigma(v)) : \overline{A}(V_1) \to A(sort(v)) \mid v \in V_2\rangle$, $x \in A(V_1)$, *the mediating morphism for the total functions* $A(\sigma(v))$.

Proof. It is obvious that $\overline{A}$ preserves composition and identities. □

With this functor we may consider an arbitrary algorithm as the definition of a total set-theoretic function. This is our basis for understanding the semantics of programs.

Definition 11 (Equivalent algorithms). *Given a signature* Σ *and an algebra* $A : \Sigma \to \mathbf{Set}$, *the set* $\mathcal{A}(A, a : V_1 \to V_2) = \{a' : V_1 \to V_2 \mid \overline{A}(a') = \overline{A}(a)\}$ *is the collection of all algorithms* a' *equivalent to a given algorithm* a *in the algebra* A *for* Σ.

3.2 Assignment Program Category $\mathbf{Prog}_{:=}(\Theta)$

Now we will describe different program category, the category $\mathbf{Prog}_{:=}(\Theta)$, that more closely resembles an imperative programming language.

Given an environment, we may assign a term to one variable, create new variable (and assign a term to it) or we may delete a variable. The last two operations change a variable set we are working on.

Definition 12 (Instructions with environment). *The class of instructions with environment over signature $\Theta = (S, F)$, denoted $IE(\Theta)$, is defined as follows:*

- $\langle V, V, x := t\rangle \in IE(\Theta)$ *when* $x \in V$, $t \in T(\Theta, V)$ *and* $sort_V(x) = tsort(t)$,
- $\langle V, V \cup \{x\}, new\ s\ x := t\rangle \in IE(\Theta)$ *when* $x \notin V$, $t \in T(\Theta, V)$ *and* $s = tsort(t) = sort_{\{V\cup\{x\}\}}(x)$,
- $\langle V, V\ -\{x\}, del\ x\rangle \in IE(\Theta)$ *when* $x \in V$.

In general, an element of $IE(\Theta)$ will be denoted as $\langle V, V', i\rangle$. Given two variable sets V and V', a class of all $\langle V, V', i\rangle$ is denoted $IE(\Theta)_{V,V'}$

Definition 13 (Used variables). *Given signature $\Theta = (S, F)$, and a sets of variables V' and V such that $V' \subseteq V$. $ivar_V$-function of V is a map $IE(\Theta) \to Pow(V)$ defined in following way:*

- $ivar_V(\langle V', V', x := t\rangle) = tvar_V(t) \cup \{x\}$ *for any* $x \in V$
- $ivar_V(\langle V', V' \cup \{x\}, new\ s\ x :=, t\rangle) = tvar_V(t) \cup \{x\}$ *for any* $x \in V' - V$,
- $ivar_V(\langle V', V'\ -\{x\}, del\ x\rangle) = \{x\}$ *for any* $x \in V'$.

Function $ivar_V$ can be viewed as extension of corresponding $tvar_V$ function from terms to instructions with environment.

Definition 14 (Right hand side of an instruction). *We will use $rhs(i)$ to denote the term t used in instruction i:*

- $rhs(x := t) = t$
- $rhs(new\ s\ x := t) = t$
- $rhs(del\ x)$ *is undefined.*

Definition 15 (Instruction lists with environment). *A class of instruction lists over signature Θ, denoted $IL(\Theta)$ is defined inductively as follows:*

- $\langle V, V, \epsilon\rangle \in IL(\Theta)$ *for every set of variables V (empty set included).*
- $i \in IE(\Theta) \Rightarrow i \in IL(\Theta)$
- $\langle V_1, V_2, l\rangle, \langle V_2, V_3, l'\rangle \in IL(\Theta) \Rightarrow \langle V_1, V_3, l; l'\rangle \in IL(\Theta)$

Class $IL(\Theta)$ is the closure of the class $IE(\Theta)$ under concatenation. In general, a member of $IL(\Theta)$ will be denoted as a triple $\langle V_1, V_2, l\rangle$. Further in the text, we will use $\mathbf{x_i} := \mathbf{t_{i(V)}}$ as a short version for a list of assignments $< x_1 := t_1; x_2 := t_2;\ \ldots\ x_n := t_n >$ over some variable set $V = \{x_1, x_2,\ \ldots\ x_n\}$. We will use similar notation for lists of variable creations ($\mathbf{new\ s\ x_i} := \mathbf{t_{i(V)}}$) and lists of variable destructions ($\mathbf{del\ x_{i(V)}}$).

Definition 16 (Assignment program category $\mathbf{Prog}_{:=}(\Theta)$). *Given signature $\Theta = (S, F)$. A category $\mathbf{Prog}_{:=}(\Theta)$ is given by*

- *Objects: a* data structure *V is a set of variables V over Θ.*
- *Morphisms: an* algorithm *$a : V_1 \to V_2$ is an element (denoted by $\langle V_1, V_2, l_a \rangle$) of $IL(\Theta)$.*
- *Composition: list concatenation*
- *Identity: $\langle V, V, \epsilon \rangle$ is identity morphism for any $V \in |\mathbf{Prog}_{:=}(\Theta)|$.*

This gives an associative composition where the identity morphism for object V is given by the triple $\langle V, V, \epsilon \rangle$.

Example 17. Given the signature $\Theta = (\{int\}, \{+ : int, int \to int\})$, the triple $\langle \{x, y\}, \{z\}, new\ s\ z := x * y + x * y\ ;\ del\ x\ ;\ del\ y \rangle$ is one possible morphism between objects $\{x, y\}$ and $\{z\}$ in $\mathbf{Prog}_{:=}(\Theta)$.

Example 18. Given the signature $\Theta = (\{int\}, \{+ : int, int \to int\})$, the triple $\langle \{x, y\}, \{z\}, new\ s\ w := x * y\ ;\ new\ s\ z := w + w\ ;\ del\ w\ ;\ del\ x\ ;\ del\ y \rangle$ is another possible morphism between objects $\{x, y\}$ and $\{z\}$ in $\mathbf{Prog}_{:=}(\Theta)$.

Notice that these two different morphisms of $\mathbf{Prog}_{:=}(\Theta)$ are **Set**-semantically related to the morphism introduced in the example 5.

Since all syntactically different morphisms are different, it is easy to prove that category does not have products, coproducts, equalizers and coequalizers.

By assigning a runtime-cost to variable creations, variable assignments, variable deletions and to each operation in Σ, we are able to differentiate between runtime costs of the morphisms in $\mathbf{Prog}_{:=}(\Theta)$. For example, allocation and deallocation of variables in allows us to write code that takes runtime into consideration - we may write $new\ s\ x := t\ ;\ y := t'[t \to x]\ ;\ del\ x$ instead of $y := t'$ when t occurs several times in t'.

3.3 Quotiented Assignment Program Category

We may introduce some equivalence of morphisms in $Prog_{:=}(\Theta)$. The equivalence is given by the following relation:

Definition 19 (Equivalence of morphisms in $Prog_{:=}(\Theta)$ - relation $\simeq$). *The following instruction lists are $\sim_b$-related:*

- *elimination rules:*

$$\begin{array}{|lll|l|}
\hline
\langle V, V, x := t\ ;\ x := t' \rangle & \sim_b & \langle V, V, x := t'[x \to t] \rangle, & (1el) \\
\langle V, V', x := t\ ;\ del\ x \rangle & \sim_b & \langle V, V', del\ x \rangle, & (2el) \\
\langle V, V', new\ s\ x := t\ ;\ x := t' \rangle & \sim_b & \langle V, V, new\ s\ x := t'[x \to t] \rangle, & (3el) \\
\langle V, V, new\ s\ x := t\ ;\ del\ x \rangle & \sim_b & \langle V, V, \epsilon \rangle, & (4el) \\
\langle V, V, del\ x\ ;\ new\ s\ x := t \rangle & \sim_b & \langle V, V, x := t \rangle & (5el) \\
\hline
\end{array}$$

- *identity assignment rule:*

$$\boxed{\langle V, V, x := x \rangle \sim_b \langle V, V, \epsilon \rangle\ (1ai)}$$

– *variable substitution rules:*

$$\begin{array}{lll} & \langle V, V, i[x{\rightarrow}x]\rangle{\sim_b} & \\ & \langle V, V, i\rangle & (1vs) \\ y \notin ivar(i) \wedge z \notin ivar(i[x{\rightarrow}y]) \Rightarrow & \langle V, V, i[x{\rightarrow}y][y{\rightarrow}z]\rangle{\sim_b} & \\ & \langle V, V, i[x{\rightarrow}z]\rangle & (2vs) \\ y \notin ivar(i) \Rightarrow & \langle V, V, new\ s\ y := x\ ;\ i\rangle{\sim_b} & \\ & \langle V, V, i[y{\rightarrow}x]\ ;\ new\ s\ x := y\rangle & (3vs) \end{array}$$

– *change order of instruction rules. In these rules, i_x denotes any instruction that has the variable x on the left side of the instruction (including del (x)). Similarly, i_y denotes any instruction that has the variable y on the left side of the instruction:* [3]

$$\begin{array}{l} (x \neq y) \wedge (y \notin ivar(i_x)) \Rightarrow \\ \quad \langle V, V', i_x\ ;\ i_y\rangle{\sim_b}\langle V, V', i_y[x{\rightarrow}rhs(i_x)]\ ;\ i_x\rangle \qquad (1ie) \\ (x \neq y) \wedge (y \in ivar(i_x)) \Rightarrow \\ \quad \langle V, V', i_x\ ;\ i_y\rangle{\sim_b}\langle V, V', new\ sort(y)\ y' := y\ ;\ i_y[x{\rightarrow}rhs(i_x)]\ ;\ i_x[y{\rightarrow}y']\ ;\ del\ y'\rangle \end{array}$$

Relation $\simeq\ \subseteq IL \times IL$ is the smallest equivalence relation that is the reflexive, transitive and symmetric closure of relation $\sim_b$, closed under concatenation: whenever $(a_1, a_2) \in \simeq$ and $(b_1, b_2) \in \simeq$, then $(a_1; b_1, a_2; b_2) \in \simeq$. Further in the text, the $\simeq$-class that contain instruction list $\langle V_1, V_2, a\rangle$, will be denoted $[\langle V_1, V_2, a\rangle]_\simeq$ or just $[\langle V_1, V_2, a\rangle]$.

Definition 20. *Category* $\mathbf{Prog}_{:=/\simeq}(\Theta)$ *is a category with*

– *Objects: are objects of* $\mathbf{Prog}_{:=}(\Theta)$.
– *Morphisms: $a : V_1{\rightarrow}V_2$ are $\simeq$-equivalence classes.*
– *Composition: the composition of two equivalence classes $[\langle V_1, V_2, a_1\rangle]$ and $[\langle V_2, V_3, a_2\rangle]$ is equivalence class $[\langle V_1, V_3, a_1; a_2\rangle]$.*
– *Identity: equivalence class that contains $\langle V, V, \epsilon\rangle$ is identity morphism for any $V \in |\mathbf{Prog}_{:=}(\Theta)|$.*

Notice that the morphisms introduced in the examples 17 and 18 are equivalent in $\mathbf{Prog}_{:=/\simeq}(\Theta)$.

Definition 21. *Functor* $TA : \mathbf{Prog}_T(\Theta) \rightarrow \mathbf{Prog}_{:=/\simeq}(\Theta)$ *is given by*

– $TA(V) = V$,
– $TA(< V_1, V_2, \sigma_a : V_2 \rightarrow T(\Theta, V_1) >) =< V_1, V_2, l >$ *where*

$$\begin{array}{rl} l = < & \mathbf{x_i} := \sigma_{\mathbf{a}}(\mathbf{x_i})_{(\mathbf{V_2} \cap \mathbf{V_1})}; \\ & \mathbf{new\ sort}(\mathbf{x_j})\ \mathbf{x_j} := \sigma_{\mathbf{a}}(\mathbf{x_j})_{(\mathbf{V_2} - \mathbf{V_1})}; \\ & \mathbf{del}\ \mathbf{x_{k(V_1 - V_2)}}; \\ > & \end{array}$$

for all $x_i \in V_2 \cup V_1$, all $x_j \in V_2/V_1$ and all $x_k \in V_1/V_2$.

[3] Notice that in the 'change order of instruction' rules we accept the possibility that substitution has no effect (when there is nothing to substitute, such as (1ie) when both instructions are assignments and $x \notin tvar(rhs(i_y))$ or, such as (1ie) when rhs is undefined - when one or both instructions are 'delete sentences'. In addition, rules do not apply for impossible combinations (such as $y \in ivar(i_x)$ with $i_x = insdx$).

In order to define a functor from $\mathbf{Prog}_{:=/\simeq}(\Theta)$ to $\mathbf{Prog}_T(\Theta)$, notice that for every instruction list with environment $< V_1, V_2, l >$, by applying the equivalence rules, we may build an equivalent instruction list with environment $< V_1, V_2, l_c >$ where l_c is such that:

- all temporary variables (not belonging to $V1$ nor $V2$) are removed,
- for every variable $v_i \in V_2$, there exist only one instruction where v_i appears on the left side of an expression (either $v_i := t_i$ or *new sort*(v_i) $v_i := t_i$,
- that have **del** $\mathbf{v_{j(v_j \in V_1 - V_2)}}$ at the end of the list, and
- that have no other instruction.

Definition 22. *Functor* $AT : \mathbf{Prog}_{:=/\simeq}(\Theta) \to \mathbf{Prog}_T(\Theta)$ *is given by*

- $AT(V) = V$,
- $AT(\langle V_1, V_2, l\rangle) = \langle V_1, V_2, \sigma_a : V_2 \to T(\Theta, V_1)\rangle$ *where substitution* $\sigma_a(v) = t$ *for a variable* $v \in V_2$ *and an appropriate term from* $< V_1, V_2, l_c >$.

There are two main differences between program categories $\mathbf{Prog}_T(\Theta)$ and $\mathbf{Prog}_{:=}(\Theta)$. One is variable management in $\mathbf{Prog}_{:=}(\Theta)$ and the other is order of instruction execution. Booth of these differences disappear when we consider $\mathbf{Prog}_{:=/\simeq}(\Theta)$ due to equivalence rules.

Theorem 23. *Category* $\mathbf{Prog}_T(\Theta)$ *is isomorphic to category* $\mathbf{Prog}_{:=/\simeq}(\Theta)$.

For the proof we refer to [Hod10]. The proof is done by showing that AT and TA are functors, and that compositions $AT; TA$ and $TA; AT$ are identity functors.

This theorem implies that categorical properties of $\mathbf{Prog}_T(\Theta)$ are also categorical properties of $\mathbf{Prog}_{:=/\simeq}(\Theta)$.

3.4 Implementing an Interface Σ in $\mathbf{Prog}_T(\Theta)$

When we implement a suite of application programs or implement a new package, we will provide a code for some interface, which we may consider a total many-sorted signature Σ. In our framework the implementation will be data structures and algorithms in $\mathbf{Prog}_T(\Theta)$, where Θ represents the signature of the libraries and built-in types and operations. We place no apriory restrictions on the relationship between Θ and Σ. If we are developing a library package, it may be natural to assume that $\Theta \subseteq \Sigma$ and require that the implementation of Θ remains unchanged in the development of the rest of Σ. But such a requirement would make it impossible to re-implement an existing library package, e.g., replacing an inefficient symbol table by a more efficient one. Likewise, if we are creating application programs, we may want to expose only a minor part of Θ in the interface. Consider the example of a collection of application programs for a data base system. Here we do not want to expose all the library packages that are used in its implementation in the interface of the application software.

By not placing any restrictions on Θ and Σ we are free to exploit such situations in our framework. However, adding some such restrictions at a later stage

may provide additional information for certain kinds of software development situations.

We now define what it means to implement a signature Σ by algorithms in Θ in a programming language given by the category $\mathbf{Prog}_T(\Theta)$.

Definition 24 (Implementation in $\mathbf{Prog}_T(\Theta)$). *Given signatures Σ and Θ. The* implementation $I_T : \Sigma \to \mathbf{Prog}_T(\Theta)$ *of Σ by algorithms in Θ defines*

- *a data structure $I_T(s) \in |\mathbf{Prog}_T(\Theta)|$ for every sort $s \in S$,*
- *a product data structure $\Pi_{r \in R} I_T(r)$ for every family R of S,*
- *an algorithm $(I_T(f) : I_T(\mathrm{dom}(f)) \to I_T(\mathrm{cod}(f))) \in \mathtt{Mor}(\mathbf{Prog}_T(\Theta))$ for every $f \in F$.*

Example 25. Given the signatures $\Theta = (\{int\}, \{+ : int, int \to int\})$ and $\Sigma = (\{s_1, s_2\}, \{o : s_1 \to s_2\})$, mapping $I(s_1) = \{x, y\}$, $I(s_2) = \{z\}$ and $I(o) = \langle \{x, y\}, \{z\}, \sigma(z) = x * y + x * y \rangle$ is an implementation of Σ in $\mathbf{Prog}_T(\Theta)$.

We will also use I_T to translate an arbitrary data structure X (collection of variables) on Σ to a data structure in Θ, by defining $I_T(X) = \Pi_{v \in X} I_T(\mathrm{sort}(v))$.

Definition 26. *Given signatures $\Sigma = (S, F)$ and Θ with variables V, and an implementation $I_T : \Sigma \to \mathbf{Prog}_T(\Theta)$.*
The application $I_T(t) : I_T(V) \to I_T(s)$ of I_T to terms $t \in T(\Sigma, V)_s$, $s \in S$, is defined by

- *for a compound $t = f(\Pi_{r \in \mathrm{dom}(f)} t_r)$ with $t_r \in T(\Theta, X)$, $r \in \mathrm{dom}(f)$, and $f \in F$, we have $I_T(t) = \langle I_T(t_r) \mid r \in \mathrm{dom}(f) \rangle ; I_T(\langle f \rangle)$, and*
- *for a variable $t \in V$, we have $I_T(t) = \langle I_T(s), I_T(s), \sigma : I_T(s) \to T(\Sigma, I_T(s)) \rangle$, where $\sigma(v) = v$ for all $v \in I_T(s)$.*

We may extend an implementation I to a functor $\overline{I}$ between the Θ algorithms and the Σ algorithms.

Proposition 27 (Implementations in $\mathbf{Prog}_T(\Theta)$ as functors). *Given signatures Σ and Θ. An implementation $I_T : \Sigma \to \mathbf{Prog}_T(\Theta)$ for signature Σ extends to a functor $\overline{I_T} : \mathbf{Prog}_T(\Sigma) \to \mathbf{Prog}_T(\Theta)$ given by*

- *$\overline{I_T}(V) = \Pi_{v \in V} I_T(sort(v))$ for every set of variables V over Σ.*
- *$\overline{I_T}(a : V_1 \to V_2) = \langle I_T(\sigma(v)) : \overline{I_T}(V_1) \to I_T(sort(v)) \mid v \in V_2 \rangle$ for every morphism $a = \langle V_1, V_2, \sigma : V_2 \to T(\Theta, V_1) \rangle$.*

Proof. It is obvious that $\overline{I_T}$ preserves composition and identities. □

For every implementation I_T we get a subcategory $\overline{I_T}(\mathbf{Prog}_T(\Sigma))$ of a category $\mathbf{Prog}_T(\Theta)$.

Implementations will be models in the institution we are building. In order to build a category $\mathbf{Imp}(\Theta)$ of implementations for a signature Θ, we need to consider appropriate morphisms in such category. Since objects (implementations) may be extended to functors, it is natural to consider natural transformations as relations between the objects.

Given two implementations I, I' of Θ, an implementation morphism may be defined as the natural transformation $i : \overline{I_T} \Rightarrow \overline{I'}_T$. However, since equality in $\mathbf{Prog}_T(\Theta)$ requires equality on algorithms (which are based on syntactic terms), there are few cases where we have interesting implementation morphisms. This notion basically gives us the isomorphisms (renaming of variables) on implementations.

Definition 28 (Implementation homomorphism). *Given signatures Σ and Θ, algebra $A : \Theta \to \mathbf{Set}$ and implementations $I_T, I'_T : \Sigma \to \mathbf{Prog}_T(\Theta)$. An* implementation homomorphism $h : I_T \to I'_T$ *is a natural transformation $h : \overline{I_T}; \overline{A} \Rightarrow \overline{I'_T}; \overline{A}$.*

The more semantic implementation homomorphisms considers the meaning of an implementation as an algebra $\overline{A} \circ I_T : \Sigma \to \mathbf{Set}$, and uses the standard notion of homomorphism for total many-sorted algebras. Similar implementation (as functor) constructions may be built for program categories $\mathbf{Prog}_{:=}(\Theta)$ or $\mathbf{Prog}_{:=/\simeq}(\Theta)$.

Similar notion of implementation and implementation morphism may be defined for the two other program categories that we have considered.

3.5 A general program category $\mathbf{Prog}_L(\Theta)$ for a language L

So far, we have presented three program categories: $\mathbf{Prog}_T(\Theta)$, $\mathbf{Prog}_{:=}(\Theta)$ and $\mathbf{Prog}_{:=/\simeq}(\Theta)$. Other program categories are certainly possible.

For any given language L and a signature Θ (the library including primitive types and operations), we may build a program category. The approach presented here requires that the following is satisfied:

- Objects of program category $\mathbf{Prog}_L(\Theta)$ are datastructures and morphisms are program code/terms (algorithms).
- Given an algebra $A : \Theta \to \mathbf{Set}$, we should be able to extend it to a functor $\overline{A} : \mathbf{Prog}_L(\Theta) \to \mathbf{Set}$ mapping objects to datatypes and morphisms to functions. This means that the language being considered has **Set**-based semantics.
- There must exist a mapping $I : \Sigma \to \mathbf{Prog}_T(\Theta)$ This means that we may implement signatures as program code.
- Every mapping I should be extendable to functor $\overline{I} : \mathbf{Prog}_L(\Sigma) \to \mathbf{Prog}_L(\Theta)$ such that $\overline{I; \overline{A}} = \overline{I}; \overline{A}$. This means that every Σ term may be replaced with the code implementing Σ in the term. This is often called inlining in compiler terminology. We have this kind of inlining for categories that resemble $\mathbf{Prog}_{:=}(\Theta)$. In $\mathbf{Prog}_T(\Theta)$, this is given by substitution rules.

Note that we may use transfinite sets to deal with iteration. In principle, the category **Set** may be replaced with other semantical categories, e.g., the category of domains, according to the language semantics.

The concrete choice of a program category will impact our ability to do reasoning on resulting constructions. For example, $\mathbf{Prog}_{:=}(\Theta)$ allows us to reason about program properties such as runtime or memory consumption. On the other

hand, all syntacticly different algorithms are different morphisms in $\mathbf{Prog}_{:=}(\Theta)$, so we have no means of talking about equivalent algorithms in $\mathbf{Prog}_{:=}(\Theta)$. Category $\mathbf{Prog}_{:=/\simeq}(\Theta)$ allows us to discuss equivalence of morphisms but, due to the equivalence of algorithms, we lose ability to reason about runtime or memory consumption.

3.6 A Simple Program Institution

In order to establish a model category for an institution, the choice of $\mathbf{Prog}_L(\Theta)$ is less important since the internal structure of the model categories is not part of the requirements for an institution.

Definition 29 (Implementation category). *Let Σ and Θ be signatures and $A : \Sigma \to \mathbf{Set}$ an algebra. The* implementation category $\mathbf{Imp}_{\Theta,A}(\Sigma)$ *has implementations $I : \Sigma \to \mathbf{Prog}_L(\Theta)$ as objects and implementation homomorphisms on $\overline{A} \circ I$ as morphisms.*

Every $\mathbf{Imp}_{\Theta,A}(\Sigma)$ is a functor-category with $\mathbf{Prog}_L(\Theta)$ as its basis. This means that categorical properties of $\mathbf{Prog}_L(\Theta)$ will be preserved also in $\mathbf{Imp}_{\Theta,A}(\Sigma)$ so by implementing in $\mathbf{Prog}_L(\Theta)$ we get (co)complete $\mathbf{Imp}_{\Theta,A}(\Sigma)$. Implementing in $\mathbf{Prog}_T(\Theta)$ (or in $\mathbf{Prog}_{:=/\simeq}(\Theta)$) will result in an implementation category that have $\mathbf{Prog}_T(\Theta)$ categorical properties.

Definition 30 (Implementation retract). *Let Θ, $\Sigma = (S, F)$ and Σ' be signatures, $\theta : \Sigma \to \Sigma'$ a signature morphism, and $A : \Theta \to \mathbf{Set}$ an algebra.*
An implementation retract *is a functor $\mathbf{Imp}_{\Theta,A}(\theta) : \mathbf{Imp}_{\Theta,A}(\Sigma') \to \mathbf{Imp}_{\Theta,A}(\Sigma)$ defined by*

- $\mathbf{Imp}_{\Theta,A}(\theta)(I' : \Sigma' \to \mathbf{Prog}_L(\Theta)) = I : \Sigma \to \mathbf{Prog}_L(\Theta)$ *where $I(s) = I'(\theta(s))$ for $s \in S$ and $I(f) = I'(\theta(f))$ for $f \in F$,*
- $\mathbf{Imp}_{\Theta,A}(\theta)(h' : I' \to J') = h : I \to J$ *where $h_s = h'_{\theta(s)}$ for all $s \in S$.*

The combination of an implementation category for each signature and an implementation retract functor for each signature morphism, gives us a model functor $\mathbf{Imp}_{\Theta,A} : \mathbf{Sig}^{\mathsf{op}} \to \mathbf{CAT}$.
We derive a satisfaction relation on implementations $I : \Sigma \to \mathbf{Prog}_L(\Theta)$ from the satisfaction relation on algebras $A : \Sigma \to \mathbf{Set}$.

Definition 31 (Program satisfaction). *Let Θ and $\Sigma = (S, F)$ be signatures, and $A : \Sigma \to \mathbf{Set}$ an algebra. An implementation $I : \Sigma \to \mathbf{Prog}_L(\Theta)$* satisfies *a specification $\varphi \in Sen(\Sigma)$ for $\mathcal{TEL}$ with variables V for Σ, written $I \models_{\Theta,A,\Sigma} \varphi$, iff for all assignments $\alpha : V \to I; \overline{A}$ we have that $(I; \overline{A})(\alpha)(\varphi)$ holds.*

The important point here is that satisfaction is by all elements in the algebra for the implemented data structure, i.e., the satisfaction relation for the algebra $(I; \overline{A})$.

Proposition 32 (Program satisfaction condition). *Let Σ, Θ and Σ' be signatures, $I' : \Sigma' \to \mathbf{Prog}_L(\Theta)$ an implementation, $\theta : \Sigma \to \Sigma'$ a signature morphism, $\varphi \in Sen(\Sigma)$ a sentence, and $A : \Theta \to \mathbf{Set}$ an algebra.*
Then

$$I' \models_{\Theta,A,\Sigma'} Sen(\theta)(\varphi) \Leftrightarrow \mathbf{Imp}_{\Theta,A}(\theta)(I') \models_{\Theta,A,\Sigma} \varphi.$$

Proof. Follows from $\mathcal{TEL}$ since the satisfaction relation is the same. □

We have now proven the following theorem.

Theorem 33 (Program institution). *Let $\Sigma \in |\mathbf{Sig}|$ be a signature and $A : \Theta \to \mathbf{Set}$ an algebra.*
Then we get an institution $\mathcal{IMP}(\Sigma, A)$ where

- *total many-sorted signatures* **Sig** *is the signature category,*
- *equations $Sen : \mathbf{Sig} \to \mathbf{Set}$ are the sentences (as in $\mathcal{TEL}$)*
- *the implementation functor $\mathbf{Imp}_{\Theta,A} : \mathbf{Sig}^{op} \to \mathbf{CAT}$ is the model functor,*
- *satisfaction is given by the family $\models_{\Theta,A}$ of satisfaction relations $\models_{\Theta,A,\Sigma} |\mathbf{Imp}_{\Theta,A}(\Sigma)| \times |Sen(\Sigma)|$ for each $\Sigma \in |\mathbf{Sig}|$.*

4 Conclusion and future work

In this paper we have presented a technique for using programs as models in an institution. First, we have introduced three different program categories ($\mathbf{Prog}_T(\Theta)$, $\mathbf{Prog}_{:=}(\Theta)$ and $\mathbf{Prog}_{:=/\sim}(\Theta)$). With these, we are able to define the (general) notion of implementation. We also gain the insight into what we need to take into consideration when building a program category of interest (discussed in 3.5). Next, we have defined the concept of implementation showing how to build the libraries for a programming language. Finally, we have built an institution in which model categories are based on (syntactical) implementations.
By using a model category based on implementations instead of **Set**, we obtain the ability to differentiate between programs that are syntactically different but **Set**-semantically equal. This opens up for a possibility to study concepts like data structure, algorithm, run-time complexity, memory-usage etc. in an institutional framework. This work could be further expanded in several directions:

- Examine how we may study program properties within the framework - work on this will be presented in [Hod10].
- Examine additional, more expressive program categories. Program categories that allows us to express encapsulation of the data structure will be presented in [Hod10].
- Here, we have focused on equational specifications for total many-sorted algebras. Since our underlying model is algebraic, it seems straight forward to apply these techniques to other types of logic.
- Consider implementation between different languages, i.e. implementation mappings $\mathbf{Prog}_L(\Sigma) \to \mathbf{Prog}_{L'}(\Theta)$ which would give an insight into compilers (mapping *code* → *assembly*) and program translation in general.

References

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, 1998.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley, 2000.

[Fia04] Jose Luiz Fiadeiro. *Categories for Software Engeneering*. Springer Verlag, 2004.

[GB92] Joseph A. Goguen and R. M. Burstall. Institutions: Abstract model theory for computer science. *Journal of the Association for Computing Machinery*, 39:95–146, 1992.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Hod10] Adis Hodzic. *Software Institutions*. PhD thesis, Department of Informatics, University of Bergen, In preparation, 2010.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 – European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, 1997.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8), 1977.

[ST97] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Asp. Comput.*, 9(3):229–269, 1997.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[Wag90] Eric G. Wagner. Algebras, polynomials and programs. *Theor. Comput. Sci.*, 70(1):3–34, 1990.

[Wag91] Eric G. Wagner. Generic classes in an object-based language. In Michel Bidoit and Christine Choppy, editors, *COMPASS/ADT*, volume 655 of *Lecture Notes in Computer Science*, pages 330–344. Springer, 1991.

[Wal91] R.F.C. Walters. *Categories and Computer Science*. Number 28 in Cambridge computer science texts. Cambridge University Press; Cambridge, GB, 1991.

[Wir76] Niklaus Wirth. *Algorithms + data structures = programs*. Prentice-Hall series in automatic computation. Prentice-Hall; Englewood Cliffs, N.J., 1976.

Regaining Confluence in λ^{Gtz}-calculus

Jelena Ivetić

Faculty of Technical Sciences, University of Novi Sad, Serbia
jelena@imft.ftn.uns.ac.rs

Abstract. In this paper, we propose two confluent subcalculi of λ^{Gtz}, the intuitionistic sequent term calculus of Espírito Santo. These calculi are obtained from λ^{Gtz} by putting restrictions over the reduction rules in a way which eliminates the critical pair. We prove the Church-Rosser property of the proposed calculi using Takahashi's parallel reductions technique.

1 Introduction

Designing a term calculus which embodies the Curry-Howard correspondence for the intuitionistic sequent calculus took several attempts over the years. The first successful proposal was Herbelin's $\bar{\lambda}$ calculus introduced in 1995 [5], where the type assignment to normal forms corresponds to the derivation in the modification of the cut-free Gentzen's sequent calculus LJT_{cf}. One of the recent calculi that embodies full Curry-Howard correspondence for intuitionistic sequent calculus (with cut) is the λ^{Gtz}-calculus, proposed by Espírito Santo in [3]. Simply typed λ^{Gtz}-calculus enjoys Subject reduction and Strong normalization property. Moreover, λ^{Gtz}-calculus with intersection types, studied in [4], enjoys both Strong normalization and Characterization of strong normalization property, in other words, in that system the set of typeable terms coincides with the set of strongly normalizing terms, which is very useful property for implementation. One of the specific properties of the λ^{Gtz}-calculus is its reduction system, particularly β-reduction. Substitution is not integrated into β-reduction, like in λ-calculus, but performed separately, through σ-reduction. Furthermore, substitution is not explicit, like in λx-calculus of Rose [7], but meta-operator. This allows us to delay substitution in λ^{Gtz}-calculus, and to simulate both call-by-name and call-by-value computational strategy, as discussed in [2]. However, there is one undesired consequence of such reduction system - it contains a critical pair, which leads to non-confluence.

In this paper, we study possible ways to regain confluence. There are two basic directions in solving this problem. The first one is to make restrictions over the reduction rules in a way which would eliminate the critical pair. The other one is to expand the syntax and create an appropriate type assignment system such that the confluence is obtained for reductions performed on all well typed terms. Here, we focus only on the first approach. We propose two sub-calculi without critical pairs and prove the confluence. We use Takahashi's parallel reductions

technique, used in [8] for proving the confluence of λ-calculus, which is simpler then standard Tait and Martin-Löf's proof of the confluence of $\beta\eta$-reduction in λ-calculus.

The paper is organized as follows: Section 1 is introductory; in Section 2 we give the basics of simply typed λ^{Gtz}-calculus; in Section 3 two sub-calculi of λ^{Gtz}-calculus, namely λ_V^{Gtz}-calculus and λ_L^{Gtz}-calculus, are proposed and Church-Rosser property for introduced calculi is proved. Finally, in Section 4 we conclude and give directions for future work.

2 λ^{Gtz}-calculus

In this section, we briefly recall untyped and simply typed λ^{Gtz}-calculus. For a detailed account on λ^{Gtz} we refer the reader to [4].

The abstract syntax of λ^{Gtz} is given by:

$$\begin{array}{ll} \text{Terms} & t ::= x \mid \lambda x.t \mid tk \\ \text{Contexts} & k ::= \widehat{x}.t \mid t :: k \end{array}$$

A term (denoted by t, u, v,...) is either a variable, an abstraction or an application tk, called *cut*. A context (denoted by k, k',...) is either a *selection* $\widehat{x}.t$, in which the variable x is bound, or a context constructor $t :: k$, usually called *cons*. The terms and the contexts together are called expressions and denoted by E.

The reduction rules of λ^{Gtz} are the following:

$$\begin{array}{ll} (\beta) & (\lambda x.t)(u :: k) \to u(\widehat{x}.tk) \\ (\pi) & (tk)k' \to t(k@k') \\ (\sigma) & t(\widehat{x}.v) \to v[x := t] \\ (\mu) & \widehat{x}.xk \to k, \text{ if } x \notin k \end{array}$$

where $v[x := t]$ denotes meta-substitution defined in the usual way, and meta-operator $k@k'$, called *append*, is defined by:

$$(u :: k)@k' = u :: (k@k'); \qquad (\widehat{x}.t)@k' = \widehat{x}.tk'.$$

Normal forms of λ^{Gtz}-calculus are expressions that contain no cuts but the trivial ones.

$$\begin{array}{l} t_{nf} ::= x \mid \lambda x.t_{nf} \mid x(t_{nf} :: k_{nf}) \\ k_{nf} ::= \widehat{x}.t_{nf} \mid t_{nf} :: k_{nf} \end{array}$$

The type assignment system with simple types, called $\lambda^{\mathsf{Gtz}} \to$ is presented in the Figure 1. A *basis* Γ is a set $\{x_1 : A_1, \ldots, x_n : A_n\}$ of basic type assignments, where all term variables are different. $Dom\Gamma = \{x_1, \ldots, x_n\}$. A basis extension $\Gamma, x : A$ denotes the set $\Gamma \cup \{x : A\}$, where $x \notin Dom\Gamma$. There are two sorts of sequents in this system: $\Gamma \vdash t : A$ - for typing a term, and $\Gamma; B \vdash k : A$ - for typing a context. The special place after the semi-comma on the LHS of the sequent is called *stoup*. It contains the selected formula with which we continue computation.

Although λ^{Gtz} satisfies many good properties (like subject reduction and strong normalization of the simply typed version, characterization of strong normalization of the system with intersection types, preservation of β-SN...see [4]) it does not enjoy the confluence, unlike majority of intuitionistic formal calculi. The reason for non-confluence is the existence of a critical pair, consisting of π and σ reductions.

$$\frac{}{\Gamma, x : A \vdash x : A}\,(Ax) \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,(\to_R)$$

$$\frac{\Gamma \vdash t : A \;\; \Gamma; B \vdash k : C}{\Gamma; A \to B \vdash t :: k : C}\,(\to_L) \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma; A \vdash \widehat{x}.t : B}\,(Sel)$$

$$\frac{\Gamma \vdash t : A \;\; \Gamma; A \vdash k : B}{\Gamma \vdash tk : B}\,(Cut)$$

Figure 1: $\lambda^{\mathsf{Gtz}} \to$ - simply typed λ^{Gtz}

Example 1. The term of the form $(tk)(\widehat{x}.v)$ is both a π-redex and a σ-redex. Contracting it as a π-redex (the call-by-value option) we get $t(k@\widehat{x}.v)$. Contracting it as a σ redex (the call-by-name option) we get $v[x := tk]$, and in a lot of particular cases these two terms can not be reduced to the same normal form.

3 Confluent sub-calculi

In this section, we propose two calculi derived from λ^{Gtz}-calculus that do enjoy the confluence property. As already mentioned, our goal is to eliminate the critical pair, and we achieve it by restricting some reduction rules of λ^{Gtz}. These restrictions are made by modifications of the syntax.

Look at the previous example. If we forbid σ reduction to perform on the term $(tk)(\widehat{x}.v)$, we will get a "call-by-value" sub-calculus, denoted by λ_V^{Gtz}. The abstract syntax of λ_V^{Gtz} is the following:

$$\begin{array}{ll} \text{Values} & T ::= x \mid \lambda x.t \\ \text{Terms} & t ::= T \mid tk \\ \text{Contexts} & k ::= \widehat{x}.t \mid t :: k \end{array}$$

So, like in [1], we introduce *values* as a new syntactic category. Reduction rules of λ_V^{Gtz} are β, π, μ of λ^{Gtz} and

$$(\sigma_V)\; T(\widehat{x}.v) \to v[x := T].$$

This reduction system is forcing us to reduce the head of the cut to the value before substituting it instead of x in v, which is exactly the essence of the call-by-value computational strategy.

On the other hand, if we forbid π reduction to perform on the term $(tk)(\widehat{x}.v)$, we will get another confluent sub-calculus, denoted by λ_L^{Gtz}. The abstract syntax of λ_L^{Gtz} is the following:

$$\begin{array}{ll} \text{Terms} & t ::= x \mid \lambda x.t \mid tk \\ \text{Lists} & l ::= \widehat{x}.x \mid t :: k \\ \text{Contexts} & k ::= l \mid \widehat{x}.t \end{array}$$

Here, we have to introduce syntactic category of *lists*, that are subset of the contexts and whose form is $t_1 :: t_2 :: ... :: t_k :: \widehat{x}.x$. The trivial selection, $\widehat{x}.x$, actually represents an empty list $[\,]$, so by applying this convention we can write the upper list in the form $[t_1, t_2, ..., t_k]$. Reduction rules of λ_L^{Gtz} are β, σ, μ of λ^{Gtz} and

$$(\pi_L)\ (tk)l \to t(k@l).$$

In this reduction system, only the term of the form $(tk)(\widehat{x}.x)$ is at the same time σ-redex and (π_L)-redex, but applying each of these two reductions leads to the same result - tk, so the confluence is not broken.

3.1 The proof of the confluence

After elimination of the critical pair, we can prove the confluence of the λ_V^{Gtz}-calculus.

Definition 1 (Confluence, Church-Rosser property). *Reduction R is said to be confluent if its reflexive and transitive closure $\twoheadrightarrow_R$ satisfies so-called diamond-property i.e. if for all terms t, t_1, t_2 the following holds:*

$$\textit{if } t_1 \twoheadleftarrow t \twoheadrightarrow t_2, \textit{ then there exists } t' \textit{ such that } t_1 \twoheadrightarrow t' \twoheadleftarrow t_2.$$

The technique we use for proving this property is the parallel reductions technique, developed by Takahashi [8] and adapted by Likavec [6] for proving Church-Rosser property of the $\lambda\mu\tilde{\mu}$ sub-calculi. This approach is based on simultaneous reduction of all existing redexes in the term. In the sequel, $\to$ will denote the union of all four λ_L^{Gtz} reductions and $\twoheadrightarrow$ will denote its reflexive and transitive closure.

First, we need to introduce the notion of the parallel reductions for λ_V^{Gtz}-calculus, denoted by $\Rightarrow$ and defined inductively as follows:

Definition 2 (Parallel reductions for λ_V^{Gtz}-calculus).

$$\frac{}{x \Rightarrow x}\ (g1) \qquad \frac{t \Rightarrow t'}{\lambda x.t \Rightarrow \lambda x.t'}\ (g2) \qquad \frac{t \Rightarrow t',\ k \Rightarrow k'}{tk \Rightarrow t'k'}\ (g3)$$

$$\frac{t \Rightarrow t'}{\widehat{x}.t \Rightarrow \widehat{x}.t'}\ (g4) \qquad \frac{t \Rightarrow t',\ k \Rightarrow k'}{t :: k \Rightarrow t' :: k'}\ (g5)$$

$$\frac{t \Rightarrow t',\ u \Rightarrow u',\ k \Rightarrow k'}{(\lambda x.t)(u :: k) \Rightarrow u'\widehat{x}.(t'k')}\ (g6) \qquad \frac{T \Rightarrow T',\ t \Rightarrow t'}{T(\widehat{x}.t) \Rightarrow t'[x := T']}\ (g7)$$

$$\frac{t \Rightarrow t',\ k \Rightarrow k',\ k_1 \Rightarrow k_1'}{(tk)k_1 \Rightarrow t'(k' @ k_1')}\ (g8) \qquad \frac{k \Rightarrow k'}{\widehat{x}.xk \Rightarrow k'}\ (g9)$$

Lemma 1 (Reflexivity). *For every expression E, $E \Rightarrow E$.*

Proof. By induction on the structure of E. The basic case is covered by the rule $(g1)$ from definition 2. In all other cases, we apply IH on subexpressions of E and rules $(g2)$ - $(g5)$. □

Lemma 2 (Substitution). *If $x \neq y$ and $x \notin FV(v_2)$ then $E[x := v_1][y := v_2] = E[y := v_2][x := v_1[y := v_2]]$.*

Proof. By induction on the structure of E. The basic case is when E is the variable. There are three possibilities:

- $E \equiv x$.
  Then, $x[x := v_1][y := v_2] = v_1[y := v_2]$ and $x[y := v_2][x := v_1[y := v_2]]$
  $= x[x := v_1[y := v_2]] = v_1[y := v_2]$.
- $E \equiv y$.
  Then, $y[x := v_1][y := v_2] = y[y := v_2] = v_2$ and $y[y := v_2][x := v_1[y := v_2]]$
  $= v_2[x := v_1[y := v_2]] = v_2$, because $x \notin FV(v_2)$.
- $E \equiv z$, such that $z \neq x$ and $z \neq y$.
  Then both sides are equal to z.

In all other cases, we apply IH on subexpressions of E. □

Now we give the definition of *holes*.

Definition 3 (Holes).

$$\mathcal{C}_t ::= [\,] \mid \lambda x.\mathcal{C}_t \mid t\mathcal{C}_c \mid \mathcal{C}_t k$$
$$\mathcal{C}_c ::= \widehat{x}.\mathcal{C}_t \mid t :: \mathcal{C}_c \mid \mathcal{C}_t :: k$$

We will write $\mathcal{C}$ instead $\mathcal{C}_t \cup \mathcal{C}_c$. $\mathcal{C}[G]$ denotes filling the hole in C with an expression G. Now we can prove the following statement.

Lemma 3.

(i) *If $E \to E'$ then $E \Rightarrow E'$.*
(ii) *If $E \Rightarrow E'$ then $E \twoheadrightarrow E'$.*
(iii) *If $E \Rightarrow E'$ and $H \Rightarrow H'$, then $E[x := H] \Rightarrow E'[x := H']$.*

Proof. (i) By induction on the sort of the hole in redex. If $E \to E'$ then $E = \mathcal{C}[H]$, $E' = \mathcal{C}[H']$ and $H \to H'$.

- The basic case is $\mathcal{C} \equiv [\,]$.
  There are four possible cases of reductions in $H \to H'$.
  β-reduction - then $H \equiv (\lambda x.t)(u :: k)$ and $H' \equiv u(\widehat{x}.tk)$. By lemma 1 $t \Rightarrow t$, $u \Rightarrow u$ and $k \Rightarrow k$, so from rule $(g6)$ of definition 2 $H \Rightarrow H'$, hence $E \Rightarrow E'$.
  σ-reduction - then $H \equiv T\widehat{x}.v$ and $H' \equiv v[x := T]$. By lemma 1 $T \Rightarrow T$ and $v \Rightarrow v$, so from rule $(g7)$ we have that $H \Rightarrow H'$.
  π-reduction - then $H \equiv (tk)k'$ and $H' \equiv tk@k'$. By lemma 1 $t \Rightarrow t$, $k \Rightarrow k$ and $k' \Rightarrow k'$, so from rule $(g8)$ $H \Rightarrow H'$.
  μ-reduction - then $H \equiv \widehat{x}.xk$ and $H' \equiv k$. By lemma 1 $k \Rightarrow k$, so from rule $(g9)$ $H \Rightarrow H'$.

- $\mathcal{C} \equiv \lambda x.\mathcal{C}'$.
  Then $E \equiv \lambda x.\mathcal{C}'[H]$ and $E' \equiv \lambda x.\mathcal{C}'[H']$. By IH we have $\mathcal{C}'[H] \Rightarrow \mathcal{C}'[H']$, so from rule $(g2)$ of definition 2 we get $E \Rightarrow E'$.

- $\mathcal{C} \equiv t\,\mathcal{C}'$.
  Then $E \equiv t\,\mathcal{C}'[H]$ and $E' \equiv t\,\mathcal{C}'[H']$. By IH we have $\mathcal{C}'[H] \Rightarrow \mathcal{C}'[H']$, from lemma 1 we have $t \Rightarrow t$ so from rule $(g3)$ of definition 2 we get $E \Rightarrow E'$.
  The proof is similar for the rest of sorts of the hole.

(ii) By induction on the definition of the parallel reduction. We show several cases, the others are proved similarly.

- Basic case is $E \equiv x \Rightarrow x \equiv E'$.
  In that case, $E \equiv x \twoheadrightarrow x \equiv E'$ is trivially satisfied.
- $E \equiv tk \Rightarrow t'k' \equiv E'$.
  This is the direct consequence of the premises $t \Rightarrow t'$ and $k \Rightarrow k'$. By IH, $t \twoheadrightarrow t'$ and $k \twoheadrightarrow k'$, hence
  $$E \equiv tk \twoheadrightarrow t'k' \equiv E'.$$
- $E \equiv (\lambda x.t)(u :: k) \Rightarrow u'(\widehat{x}.t'k') \equiv E'$.
  This is the direct consequence of the premises $t \Rightarrow t'$, $u \Rightarrow u'$ and $k \Rightarrow k'$. By IH, $t \twoheadrightarrow t'$, $u \twoheadrightarrow u'$ and $k \twoheadrightarrow k'$, hence
  $$E \equiv (\lambda x.t)(u :: k) \to u(\widehat{x}.tk) \twoheadrightarrow u'(\widehat{x}.t'k') \equiv E'.$$

(iii) By induction on the definition of the parallel reduction.

- The first basic case is $E \equiv x \Rightarrow x \equiv E'$.
  Then $E[x := H] \equiv x[x := H] = H \Rightarrow H' = x[x := H'] \equiv E'[x := H']$, which is given in assumptions.
- The second basic case is $E \equiv y \Rightarrow y \equiv E'$, $y \neq x$.
  Then $E[x := H] \equiv y[x := H] = y \Rightarrow y = y[x := H'] \equiv E'[x := H']$, by rule $(g1)$ of definition 2.
- $E \equiv \lambda y.t \Rightarrow \lambda y.t' \equiv E'$.
  This follows from the premise $t \Rightarrow t'$, so applying IH we get $t[x := H] \Rightarrow t'[x := H']$. From this, by rule $(g2)$ and the substitution definition we get
  $$E[x := H] \equiv \lambda y.t[x := H] \Rightarrow \lambda y.t'[x := H'] \equiv E'[x := H'].$$

- $E \equiv tk \Rightarrow t'k' \equiv E'$.
  Premises of this statement are $t \Rightarrow t'$ and $k \Rightarrow k'$. Applying IH on both of them we get $t[x := H] \Rightarrow t'[x := H']$ and $k[x := H] \Rightarrow k'[x := H']$. Now we derive
  $$\begin{aligned} E[x := H] &\equiv (tk)[x := H] = t[x := H]k[x := H] \\ &\Rightarrow t'[x := H']k'[x := H'] = (t'k')[x := H'] \\ &\equiv E'[x := H'], \end{aligned}$$
  using the substitution definition and rule $(g3)$ of definition 2.
- $E \equiv \widehat{y}.t \Rightarrow \widehat{y}.t' \equiv E'$.
  The direct premise of the statement is $t \Rightarrow t'$, so applying IH on it we get $t[x := H] \Rightarrow t'[x := H']$. From this, by rule $(g4)$ and the substitution definition follows
  $$E[x := H] \equiv \widehat{y}.t[x := H] \Rightarrow \widehat{y}.t'[x := H'] \equiv E'[x := H'].$$
- $E \equiv t :: k \Rightarrow t' :: k' \equiv E'$.
  The statement follows from premises $t \Rightarrow t'$ and $k \Rightarrow k'$. Applying IH on both of them we get $t[x := H] \Rightarrow t'[x := H']$ and $k[x := H] \Rightarrow k'[x := H']$, yielding
  $$\begin{aligned} E[x := H] &\equiv (t :: k)[x := H] \\ &= t[x := H] :: k[x := H] \\ &\Rightarrow t'[x := H'] :: k'[x := H'] \\ &= (t' :: k')[x := H'] \\ &\equiv E'[x := H'], \end{aligned}$$
  by the substitution definition and rule $(g5)$ of definition 2.
- $E \equiv (\lambda y.t)(u :: k) \Rightarrow u'\widehat{y}.(t'k') \equiv E'$.
  Direct premises of this statement are $t \Rightarrow t'$, $u \Rightarrow u'$ and $k \Rightarrow k'$. By IH we have $t[x := H] \Rightarrow t'[x := H']$, $u[x := H] \Rightarrow u'[x := H']$ and $k[x := H] \Rightarrow k'[x := H']$. Now,
  $$\begin{aligned} E[x := H] &\equiv ((\lambda y.t)(u :: k))[x := H] \\ &= (\lambda y.t)[x := H](u[x := H] :: k[x := H]) \\ &\Rightarrow u'[x := H']\widehat{y}.(t'[x := H']k'[x := H']) \\ &\equiv E'[x := H'], \end{aligned}$$
  using rule $(g6)$ of definition 2.
- $E \equiv T\widehat{y}.u \Rightarrow u'[y := T'] \equiv E'$.
  This is the consequence of the premises $t \Rightarrow t'$ and $u \Rightarrow u'$. By IH we get $t[x := H] \Rightarrow t'[x := H']$ and $u[x := H] \Rightarrow u'[x := H']$, so applying lemma 2 we derive
  $$\begin{aligned} E[x := H] &\equiv (T\widehat{y}.u)[x := H] \\ &= T[x := H]\widehat{y}.u[x := H] \\ &\Rightarrow u'[x := H'][y := T'[x := H']] \\ &= u'[y := T'][x := H'] \\ &\equiv E'[x := H'], \end{aligned}$$
  using rule $(g7)$ of definition 2.

- $E \equiv (tk)k_1 \Rightarrow t'(k'@k'_1) \equiv E'$.
  The statement follows from the premises $t \Rightarrow t'$, $k \Rightarrow k'$ and $k_1 \Rightarrow k'_1$. By IH we get $t[x := H] \Rightarrow t'[x := H']$, $k[x := H] \Rightarrow k'[x := H']$ and $k_1[x := H] \Rightarrow k'_1[x := H']$. Now, from rule $(g8)$ of definition 2, we conclude

$$\begin{aligned} G[x := H] &\equiv ((tk)k_1)[x := H] \\ &= (t[x := H]k[x := H])k_1[x := H] \\ &\Rightarrow t'[x := H'](k'[x := H']@k'_1[x := H']) \\ &\equiv G'[x := H']. \end{aligned}$$

- $E \equiv \widehat{x}.xk \Rightarrow k' \equiv E'$.
  The statement follows from the premise $k \Rightarrow k'$. By IH we get $k[x := H] \Rightarrow k'[x := H']$, and then from rule $(g9)$ of definition 2 we conclude

$$\begin{aligned} E[x := H] &\equiv (\widehat{x}.xk)[x := H] \\ &= \widehat{x}.xk[x := H] \\ &\Rightarrow k'[x := H'] \\ &\equiv E'[x := H'], \end{aligned}$$

  so the proof is done. □

Expression E^*, introduced in the following definition, is obtained from E by simultaneous reducing of all existing redexes of E.

Definition 4. *Expression E^* is inductively defined as follows:*

$(*1)$ $x^* \equiv x$
$(*2)$ $(\lambda x.t)^* \equiv \lambda x.t^*$
$(*3)$ $(\widehat{x}.t)^* \equiv \widehat{x}.t^*$
$(*4)$ $(t :: k)^* \equiv t^* :: k^*$
$(*5)$ $(tk)^* \equiv t^*k^*$ *if* $tk \neq (\lambda x.v)(u :: k_1)$ *and* $tk \neq T(\widehat{x}.v)$ *and* $tk \neq (uk_1)k_2$
$(*6)$ $((\lambda x.t)(u :: k))^* \equiv u^*(\widehat{x}.t^*k^*)$
$(*7)$ $(T(\widehat{x}.v))^* \equiv v^*[x := T^*]$
$(*8)$ $((tk)k_1)^* \equiv t^*(k^*@k_1^*)$.

Theorem 1 (Star-property of $\Rightarrow$). *If $E \Rightarrow E'$, then $E' \Rightarrow E^*$.*

Proof. By induction of the structure of E. We are proving several cases, the rest are similar.

- $E \equiv x$.
  E can be only reduced in parallel to itself ie. $E' \equiv x$. On the other hand, $E^* \equiv x$.
- $E \equiv \widehat{x}.t$.
  Then $E' \equiv \widehat{x}.t'$ for some t' such that $t \Rightarrow t'$. By IH, we get $t' \Rightarrow t^*$, yielding $E' \equiv \widehat{x}.t' \Rightarrow \widehat{x}.t^* \equiv E^*$.
- $E \equiv tk$ and $E \neq (\lambda x.v)(u :: k)$ and $E \neq T(\widehat{x}.v)$ and $E \neq (uk_1)k_2$.
  In that case, applying $(g7)$ we get $E \Rightarrow t'k'$ for some t' and k' such that $t \Rightarrow t'$ and $k \Rightarrow k'$. By IH, $t' \Rightarrow t^*$ and $k' \Rightarrow k^*$, so we have $E' \equiv t'k' \Rightarrow t^*k^* \equiv E^*$.

- $E \equiv (\lambda x.t)(u :: k)$.
  Now there are two options for the structure of E', because E can be reduced in parallel by rules $(g3)$ and $(g6)$.
  1. $E' \equiv (\lambda x.t')(u' :: k')$ for some t', u' and k' such that $t \Rightarrow t'$, $u \Rightarrow u'$ and $k \Rightarrow k'$. Applying IH on each of the three subexpressions, we get $t' \Rightarrow t^*$, $u' \Rightarrow u^*$ and $k' \Rightarrow k^*$. Finally, from $(*6)$ we get $E' \equiv (\lambda x.t')(u' :: k') \Rightarrow u^* \widehat{x}.t^* k^* \equiv E^*$.
  2. $E' = u' \widehat{x}.t' k'$ for some t', u' and k' such that $t \Rightarrow t'$, $u \Rightarrow u'$ and $k \Rightarrow k'$. Applying IH on each of the three subexpressions, we again get $t' \Rightarrow t^*$, $u' \Rightarrow u^*$ and $k' \Rightarrow k^*$. The statement now holds from rules $(*3)$ and $(*5)$.
- $G \equiv T\widehat{x}.v$.
  Also in this case there are two options for the structure of E', because E can be reduced in parallel by rules $(g3)$ and $(g7)$.
  1. $E' \equiv T'\widehat{x}.v'$ for some T' and v' such that $T \Rightarrow T'$ and $v \Rightarrow v'$. By IH on both subexpressions we get $T' \Rightarrow T^*$ and $v' \Rightarrow v^*$. Finally, from $(*7)$ we get $E' \equiv T'\widehat{x}.v' \Rightarrow v^*[x := T^*] \equiv E^*$.
  2. $E' = v'[x := T']$ for some T' and v' such that $T \Rightarrow T'$ and $v \Rightarrow v'$. By IH on both subexpressions we get $T' \Rightarrow T^*$ and $v' \Rightarrow v^*$. Proposition 3 yields to $E' \equiv v'[x := T'] \Rightarrow v^*[x := T^*] \equiv E^*$. □

Now, it is easy to prove diamond-property for $\Rightarrow$.

Theorem 2 (Diamond-property for $\Rightarrow$).
If $E_1 \Leftarrow E \Rightarrow E_2$ then $E_1 \Rightarrow E' \Leftarrow E_2$ for some E'.

Finally, as a consequence of the previous Theorem 2, we obtain confluence of the λ_V^{Gtz}-calculus.

Theorem 3 (Confluence of the λ_V^{Gtz}-calculus).
If $E_1 \twoheadleftarrow E \twoheadrightarrow E_2$ then $E_1 \twoheadrightarrow E' \twoheadleftarrow E_2$ for some E'.

Remark 1. When definitions of the parallel reductions, holes and E^* are changed in accordance to the syntax and reductions rules of λ_L^{Gtz}-calculus, the proof of the confluence of the other λ^{Gtz} sub-calculus is analogous to the one above.

4 Discussion and future work

If we compare λ^{Gtz}-calculus with λ_V^{Gtz}-calculus and λ_L^{Gtz}-calculus, it is easy to observe several things. First, the sets of λ^{Gtz}-terms, λ_V^{Gtz}-terms and λ_L^{Gtz}-terms are equal. Second, simple types can be assigned to both new calculi with the typing rules that coincide with the rules of simply typed λ^{Gtz}, presented in Figure 1. It means that the sets of typeable terms in λ^{Gtz}, λ_V^{Gtz} and λ_L^{Gtz} are equal, moreover, each term has the same type in all three calculi. Third, the sets of normal forms are also equal in all three calculi, since there isn't any term that is reducible in one of the calculi, and not so in the other two (only the number of reduction paths differs). Having in mind these facts, it is obvious that the subject reduction and

the strong normalization properties of simply typed λ^{Gtz}-calculus are preserved in simply typed λ_V^{Gtz} and λ_L^{Gtz} calculi.

Introducing intersection types and searching for the way of characterizing strongly normalizing terms is still in the domain of future work, as well as exploring relations between λ_V^{Gtz}-calculus and λ_L^{Gtz}-calculus on the one side, and well known term calculi like "call-by-value" λ-calculus or confluent sub-calculi of $\lambda\mu\tilde{\mu}$-calculus on the other side.

Acknowledgments: I would like to thank Silvia Ghilezan, Jose Espírito Santo and unknown referees, for many valuable suggestions and comments; and Hugo Herbelin, for fruitful discussion that initiated this research.

References

1. Curien, P-L. and Herbelin, H.: The duality of computation. In *Proc. of the 5th ACM SIGPLAN Int. Conference on Functional Programming, ICFP 00*, pages 233243, Montreal, Canada, 2000. ACM Press.
2. Espírito Santo, J.:Delayed substitutions. In: Baader,F.(ed):*Lecture Notes in Computer Science - Proceedings of RTA'07*. Vol. 4533. Springer-Verlag (2007).
3. Espírito Santo, J.:Completing Herbelins programme. In S. Ronchi Della Rocca, editor, Proceedings of TLCA07, volume 4583 of LNCS, pages 118132. Springer-Verlag, 2007.
4. Espírito Santo, J., Ghilezan, S. and Ivetić, J.: Characterising strongly normalising intuitionistic sequent terms. In *International Workshop TYPES 07 (Selected Papers)*, volume 4941 of LNCS, pages 8599. Springer-Verlag, 2008.
5. Herbelin, H.: A lambda calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Computer Science Logic, CSL 1994*, volume 933 of LNCS, pages 6175. Springer- Verlag, 1995.
6. Likavec, S.: Types for object-oriented and functional programming languages. PhD thesis. Universita degli studi di Torino, Torino (2004).
7. Rose, K.: Explicit substitutions: Tutorial and survey. Technical Report LS-96-3, BRICS, 1996.
8. Takahashi, M.: Parallel reduction in lambda calculus. In *Information and Computation,* Vol. 118. Academic press (1995) 120–127.

SAT-based Model Checking of Train Control Systems

Phillip James* and Markus Roggenbach**

Swansea University, Swansea, Wales, UK
cspj@swansea.ac.uk, csmarkus@swansea.ac.uk

Abstract. In this paper, we demonstrate the successful application of various SAT-based model checking techniques to verify train control systems. Starting with a propositional model for a control system, we show how execution of the system can be modelled via a finite automaton. We give algorithms to perform SAT-based model checking over such an automaton. In order to tackle state space explosion we propose slicing. Finally we comment on results obtained by applying these methods to verify two real world railway interlocking systems.

1 Introduction

Formal verification of railway control software has been identified to be one of the "grand challenges" [12] of Computer Science. Various formal methods have been applied to this area, including algebraic specification, e.g., [7], process algebraic modelling and verification, e.g., [19], and also model oriented specification, where e.g., the B method has been used in order to verify part of the Paris Metro railway [8]. In partnership with Invensys, an internationally established company specialising in railway control systems, we explore various verification approaches based on SAT solving [6].

Continuing work by Kanso et al. [16] we verify interlockings of real world train stations with respect to safety conditions. Our modelling language is propositional logic, see Figure 1: The physical layout of the train station together with an abstract safety condition, e.g., 'trains are separated by at least one empty track segment', yields a concrete safety condition φ. The initial configuration of a train station is characterised by some initialisation formula I. The control program (in ladder logic, an ISO standard [1]) of the interlocking system is translated into a transition formula T. All the above translations have been automated in [16]. Using an inductive approach, namely $I(Z) \Rightarrow \varphi(Z)$ and $T(Z, Z') \wedge \varphi(Z) \Rightarrow \varphi(Z')$, Kanso et al [16] successfully verify a medium sized real world interlocking. Some of the required safety properties are automatically proven using a SAT solver [17]. However, in some cases the SAT solver produces counter examples. These take the from of one pair of states, namely interpretations of Z and Z', which violate the safety property. In the context of the interlocking under discussion, these

* Acknowledging the support of Westinghouse Rail.
** Acknowledging the support of EPSRC under the grant EP/D037212/1.

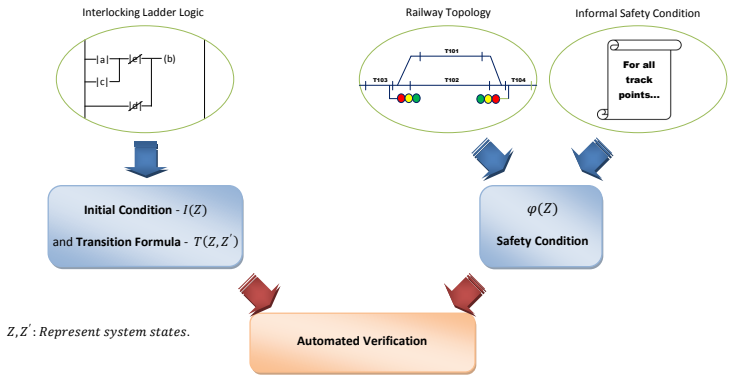


Fig. 1. The basic verification setting.

counter examples were excluded via manual analysis: it was claimed that they concern unreachable states. For inclusion into the standard development process of interlockings, Invensys requires further automation of the verification, namely

- the exclusion of unreachable states and
- the production of error traces if a safety property does not hold.

In order to accommodate these requirements, we develop and experiment with verification approaches based on ideas used in bounded model checking. Here, we deliberately stay within Boolean modelling: first, it is natural in the given context – the ladder logic program speaks on Boolean variables only; second, it allows the direct use of SAT solvers for verification.

In order to deal with real world interlockings, we develop a slicing technique. To this end we re-use an algorithm first stated by [11] and prove that it is correct w.r.t. our specific setting. In practice, slicing reduces the problem size by approximately a factor of five. This reduction has proven to be enough to verify, using various techniques, two interlockings of medium complexity: either the safety condition could be proven, or an error trace was produced.

In [20, 11] alternative approaches for the verification of ladder logic programs are provided. In [20] a translation form ladder logic into timed automata is defined, before using the Uppaal model checker [2] for verification. Due to state space explosion their approach is limited to "small" programs. Secondly, in [11] an inductive verification approach is taken to verify ladder logic interlockings.

This paper is organised as follows: In Section 2, we introduce the basics behind railway interlockings. Section 3 introduces a pelican crossing as a small example system. In Sections 4 and 5 we give a modelling of interlockings through propositional logic and automata. Section 6 introduces the model checking approaches we apply, with Section 7 giving a method to tackle state space explosion. Finally, Section 8 shows the results gained from the verification of two real world interlockings.

2 Interlockings

An interlocking provides a safety layer for a railway. It interfaces with both the physical track layout and the human (or computerised) controller. The controller issues requests, such as to move a point. On such a request the interlocking will determine whether it is safe for the operation to be permitted. If it is safe then the interlocking will change the physical track layout, informing the controller of the change. Whereas if it is unsafe to perform the operation the interlocking will not allow the physical track layout to be changed, and will report back to the controller that the operation has not taken place as it would yield an unsafe situation.

Here, we consider Westrace [3] interlockings. A Westrace interlocking has the typical control flow of a controller:

```
initialise
while True do
    read (Input)
(*) process (Input, State)
    write (Output) & update (State)
```

After initialisation, there is a loop which consists of three steps, which are repeated indefinitely in this order: (1) Reading of `Input`, where `Input` includes requests from signallers and data from physical track sensors. (2) Internal processing: this depends on the `Input` as well as on the current `State` of the controller. (3) Committing of `Output`, which includes passing information back to the signaller, commands to change the physical track layout, as well as an update of the `State` of the controller. The step `initialise` consists of the following three steps:

```
set_to_false (Input)
process (Input, State)
update(State)
```

First, all `Input` is set to false, then the step `process` is executed once, finally `State` is updated.

The Westrace interlocking realises this controller in hardware (cycle time of approximately 1 sec), where the steps `initialise` and `process` depend on the installed control software written in ladder logic – see below.

`Input`, `Output`, and `State` are sets of Boolean variables. The set `Output` is a subset of `State`. The `process` step depends on the current *configuration* of the controller, namely on the values of all variables in the sets `Input` and `State`.

3 Pelican crossing example

As a running example we study a pelican crossing, as it is found on many road networks throughout the world. The basic idea is that a pelican crossing allows pedestrians to safely cross a flow of traffic. To this end, a pelican crossing consists

of the following components: four traffic lights - two for pedestrians, two for the traffic, where for simplicity we assume that all these traffic lights can only show red or green. The pedestrians traffic lights emit an audio signal when they show green and have an input button which a pedestrian can press in order to request the green signal.

In order to program our system, we use the following Boolean variables, distinguished into input, output, and state variables. There is only one input variable, namely *pressed*. This variable becomes true if a pedestrian presses the button at either pedestrian light. We use the suffix g to indicate that a traffic light shows green, and the suffix r to indicate that a traffic light shows red. There are four traffic lights, namely *pla* and *plb* for pedestrians, and *tla* and *tlb* for traffic. Thus, overall there are eight output variables for lights, namely *pla_g*, *pla_r*, *plb_g*, *plb_r*, *tla_g*, *tla_r*, *tlb_g*, and *tlb_r*. When one of these variable is true, the corresponding light is on. There is one output variable *audio*. When *audio* is true then the audio signal is sounding. Finally there are two state variables, *req* which "remembers" the value of *pressed*, and *crossing* which indicates that pedestrians may cross the road.

$$
\begin{array}{l}
[\ \mathit{crossing}' \iff (\mathit{req} \wedge \neg \mathit{crossing}),\\
\ \ \mathit{req}' \iff (\mathit{pressed} \wedge \neg \mathit{req}),\\
\ \ \mathit{tla\_g}' \iff ((\neg \mathit{crossing}') \wedge (\neg \mathit{pressed} \vee \mathit{req}')),\\
\ \ \mathit{tlb\_g}' \iff ((\neg \mathit{crossing}') \wedge (\neg \mathit{pressed} \vee \mathit{req}')),\\
\ \ \mathit{tla\_r}' \iff \mathit{crossing}',\\
\ \ \mathit{tlb\_r}' \iff \mathit{crossing}',\\
\ \ \mathit{pla\_g}' \iff \mathit{crossing}',\\
\ \ \mathit{plb\_g}' \iff \mathit{crossing}',\\
\ \ \mathit{pla\_r}' \iff (\neg \mathit{crossing}'),\\
\ \ \mathit{plb\_r}' \iff (\neg \mathit{crossing}'),\\
\ \ \mathit{audio}' \iff \mathit{crossing}'\ ]
\end{array}
$$

Fig. 2. A ladder logic formula.

Figure 2 presents the control program of our pelican crossing. It uses unprimed variables to store the configuration of the controller *before* the step `process`. Primed variables store the values of state variables *after* the step `process`. As only `process` alters `State` variables, we can also say: if the unprimed variables represent the configuration at `(*)`, then the primed variables represent the configuration at `(*)` in the next cycle of the control loop.

The first line of Figure 2, namely "$\mathit{crossing}' \iff (\mathit{req} \wedge \neg \mathit{crossing})$", can be read as: if there was a request *req* and in the last control cycle the pedestrians were not allowed to cross the road, then at the end of the current cycle pedestrians will be allowed to cross the road. Its second line says: In the next cycle *req* will be true if a pedestrian pressed the button before starting this cycle (indicated by *pressed*) and in the previous cycle there was no request. The remainder of the program can be read similarly.

4 Ladder logic formulae

Ladder logic is graphical programming language specified in the IEC standard 61131 [1]. Westrace interlockings are programmed with ladder logic. A ladder logic program can be equivalently translated into a subset of propositional logic. We call this subset the ladder logic formulae (see below for its definition). This translation is straight forward: it simply replaces graphical symbols by logical operators, a process which has been automated in [15][1]. For the rest of the paper we only deal with this representation in propositional logic. Figure 2 gives a concrete instance using a practical shorthand notation.

Ladder logic formulae have several underlying syntactical restrictions. These restrictions become important in the context of slicing. In order to describe their syntax we use the following notations: The function *vars* returns for a given propositional formula φ the set of propositional variables appearing in φ. We use "prime" to generate a fresh variable. $V' = \{v' \mid v \in V\}$ denotes the set of all fresh variables obtained from a set of variables V.

A ladder logic program is formulated relatively to a finite set of input variables I and a finite set of state variables C, such that $I \cap C = \emptyset$. It may also refer to primed state variables C', which represent the newly computed values within a control cycle.

Definition 1 (Ladder logic formulae). *A ladder logic formula ψ (relative to a set of input variables I and a set of state variables C) is a propositional formula*

$$\psi \equiv ((c'_1 \Leftrightarrow \psi_1) \wedge (c'_2 \Leftrightarrow \psi_2) \wedge \cdots \wedge (c'_n \Leftrightarrow \psi_n))$$

where $n \geq 0$ and the ψ_i, $1 \leq i \leq n$, are propositional formulae, such that the following conditions hold:

- *for all* $1 \leq i \leq n : c'_i \in C'$.
- *for all* $1 \leq i, j \leq n :$ *if* $i \neq j \Rightarrow c'_i \neq c'_j$.
- *for all* $1 \leq i \leq n : vars(\psi_i) \subseteq I \cup \{c'_1, \ldots c'_{i-1}\} \cup \{c_i, \ldots c_n\}$.

If $n = 0$, as usual $\psi \equiv True$. The empty program proves to be useful in the context of slicing.

A ladder logic program prescribes the computation that takes place in the step `process` of the control loop. The equivalence ($\Leftrightarrow$) can be interpreted as assignment. The above conditions ensure that only primed state variables can be assigned to; a primed state variable is assigned to at most once; a primed state variable can only depend on input variables, primed state variables, or state variables – where "double use" is avoided: i.e., either the unprimed or the primed version of a state variable can be used, depending on the index i.

For a ladder logic formula we often write $\psi \equiv [R_1, R_2, \ldots, R_n]$ where $R_i \equiv c'_i \Leftrightarrow \psi_i$, for $1 \leq i \leq n$, for some $n \geq 0$. The subformulae R_i are called rungs.

[1] A similar modelling approach has been taken in [11].

5 Representation of an interlocking as an automaton

We capture the dynamics of a Westrace interlocking by defining an automaton relative to a given ladder logic formula. Consider the control loop in Section 2: a state in the automaton represents a configuration of the controller; a transition $s \to s'$ represents one execution of the loop. That is, if q represents the configuration of the controller at `(*)`, then q' represents the configuration at `(*)` one cycle later.

In order to define the transition relation via ladder logic formulae, we define paired valuations. In the definition we use I' to represent new inputs to the controller and the function *unprime* to remove the prime from a variable.

Definition 2 (Paired valuations). *Given a finite set of input variables I, a finite set of state variables C, and valuations μ, $\mu' : (I \cup C) \to \{0,1\}$ we define the paired valuation $\mu \fatsemi \mu' : (I \cup C \cup I' \cup C') \to \{0,1\}$ where*

$$\mu \fatsemi \mu'(x) = \begin{cases} \mu(x) & \text{if } x \in I \cup C \\ \mu'(unprime(x)) & \text{if } x \in I' \cup C'. \end{cases}$$

We now define an automaton for a ladder logic formula:

Definition 3 (Automaton). *Given a ladder logic formula ψ over $I \cup C$, we define the automaton*

$$A(\psi) = (S, S_0, \to)$$

where

- $S = \{\mu \mid \mu : I \cup C \to \{0,1\}\}$ *is the set of states,*
- $\mu \to \mu'$ *if* $\mu \fatsemi \mu' \models \psi$ *defines the transitions, and*
- $S_0 = \{\mu' \mid \exists \mu : \mu \models \neg I, \mu \fatsemi \mu' \models \psi\}$ *gives the set of initial states. Here, $\neg I$ expands to $\bigwedge_{i \in I} \neg i$ for all $i \in I$.*

Remark 1. The automaton $A(\psi)$ is non deterministic as ψ does not impose any conditions on the variables in I': The controller is not allowed to refuse any input. Yet another potential source of non determinism is that a state variable c' might not appear on the left hand side of the rungs of ψ. Finally, the automaton might have more than one start state. The automaton $A(\psi)$ is *finite*; it has $2^{|I \cup C|}$ states.

This automaton faithfully models the behaviour of the interlocking. The set of initial states S_0 of the automaton represents all possible configurations of the interlocking when reaching point `(*)` for the first time. As one transition corresponds to one execution of a loop, the traces of configurations observed at `(*)` directly correspond to the state sequences of the automaton.

Naturally, a controller shall never stop. In our formalisation of a Westrace interlocking we can prove this a theorem:

Theorem 1. *Let ψ be a ladder logic formula. Let μ be a state in $A(\psi)$. Then there exists a state μ' such that $\mu \fatsemi \mu' \models \psi$, i.e. it holds that $\mu \to \mu'$.*

Proof. (Sketch) By induction on size n of a ladder logic formula. Assume the claim holds for length i. Given an evaluation μ_i for $V_i = I \cup \{c'_1, \ldots, c'_{i-1}\} \cup \{c_i, \ldots, c_n\}$ we set $\mu_{i+1}(x) = \mu(x)$ for $x \in V_i$, $\mu_{i+1}(c'_{i+1}) = 1$ if $\mu_i \models \psi_{i+1}$ and $\mu_{i+1}(c'_{i+1}) = 0$ if $\mu_i \not\models \psi_{i+1}$.

We say that a paired valuation $\mu \wr \mu'$ is reachable with respect to an automaton $A(\psi) = (S, S_0, \rightarrow)$, if there exists a series of transitions $\mu_0 \rightarrow \mu_1 \rightarrow \cdots \rightarrow \mu \rightarrow \mu'$ with $\mu_0 \in S_0$.

Figure 3 illustrates the reachable states of the automaton constructed from the pelican crossing ladder logic formula in Figure 2. Here, initial states are represented via double circles, and some variable values have been excluded.

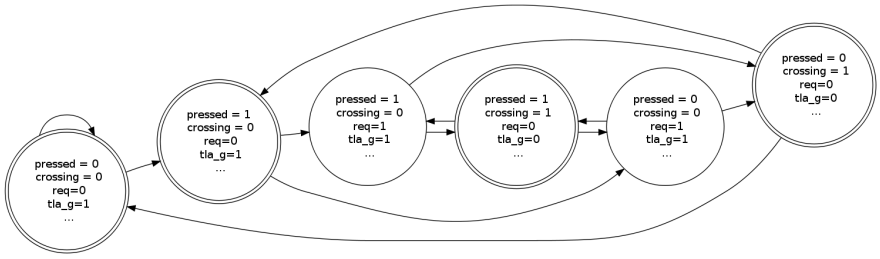


Fig. 3. An automaton modelling of the ladder logic program for a pelican crossing.

5.1 Safety conditions

A typical safety property in our pelican crossing example would be: "A traffic light always shows a single aspect". Using the vocabulary for the control program, we capture this property by the following propositional formula:

$$\mathit{SingleAspect} \equiv (\mathit{tla\_g} \vee \mathit{tla\_r}) \wedge \neg(\mathit{tla\_g} \wedge \mathit{tla\_r}) \wedge (\mathit{tlb\_g} \vee \mathit{tlb\_r}) \wedge \neg(\mathit{tlb\_g} \wedge \mathit{tlb\_r}).$$

I.e., for both traffic lights, namely *tla* and *tlb*, the following holds: either their signal is green *g* or their signal is red *r*.

Verification practice with Westrace interlockings has shown that the arising safety properties speak about at most two consecutive configurations at `(*)` of the control program depicted in Section 2. (The above example speaks only about one configuration.) This justifies the following definition:

Definition 4 (Safety condition). *A safety condition φ for a ladder logic formula ψ over variables $I \cup C$ is a propositional formula over variables $I \cup C \cup C'$.*

In this definition we exclude variables from the set I' as the controller has no effect on any input values.

5.2 The verification problem

With these notions at hand we can state our verification problem: Given a ladder logic formula ψ and a safety condition φ, we say that ψ is safe w.r.t. φ,

$$A(\psi) \models \varphi,$$

iff $\mu \fatsemi \mu' \models \varphi$ for all reachable paired valuations $\mu \fatsemi \mu'$ in $A(\psi)$.

The exclusion of non-reachable states from the verification problem is motivated by the verification results by [16] – see Section 1 – and comes as a direct request from industry. Our Pelican crossing program is safe w.r.t. *SingleAspect* only thanks to the exclusion of non-reachable states. Let μ, μ' and μ'' be states with $\mu = \{$*crossing* $= 1$, *req* $= 1$, *pressed* $= 1$, *tla_g* $= 1$, *tlb_g* $= 1$, *tla_r* $= 0$, *tlb_r* $= 0$, *pla_g* $= 0$, *plb_g* $= 0$, *pla_r* $= 1$, *plb_r* $= 1$, *audio=0*$\}$, $\mu' = \{$*crossing* $= 0$, *req* $= 0$, *pressed* $= 0$, *tla_g* $= 0$, *tlb_g* $= 0$, *tla_r* $= 0$, *tlb_r* $= 0$, *pla_g* $= 0$, *plb_g* $= 0$, *pla_r* $= 1$, *plb_r* $= 1$, *audio=0*$\}$ and μ'' any arbitrary successor of μ' (its existence is guaranteed by Theorem 1). $\mu; \mu'$ is not reachable, see Figure 3. $\mu \fatsemi \mu'$ is safe, i.e. $\mu \fatsemi \mu' \models$ *SingleAspect*, there is a transition from μ' to μ'', however, $\mu' \fatsemi \mu''$ is not safe, i.e. $\mu' \fatsemi \mu'' \not\models$ *SingleAspect*.

It is obvious how to extend our setting to safety properties that involve $k > 2$ configurations of the interlocking: instead of paired valuations one has to define k-tuples of valuations; a safety property φ can speak about k different copies of each variable in $I \cup C$; and ψ is safe if all reachable k-tuples of consecutive states satisfy the safety condition φ.

6 Applying model checking to ladder logic

In this section we discuss two verification techniques: bounded model checking [5] and temporal induction [18], both based on SAT solving. Thus, we have to give a representation of the state sequences of the automaton under consideration.

6.1 Representing state sequences

Given a set I of input variables and a set C of state variables, we define variable sets $W_i = C^{(i)} \cup I^{(i)}$ with $C^{(i)} = \{c^{(i)} \mid c \in C\}$ and $I^{(i)} = \{x^{(i)} \mid x \in I\}$ for $i \in \mathbf{Z}$. Here we use the superscript (i) to produce fresh variables. We write $[W_i/(I \cup C)]$ to denote the substitution where all superscripts are removed, and $[W_{i+1}/(I' \cup C')]$ for the substitution where all superscripts are replaced by primes. A sequence $W_0, W_1, W_2, \ldots$ of these variable sets is capable to "store" a state sequence of an automaton $A(\psi)$:

Definition 5 (Series of transitions). *Let ψ be a ladder logic formula. We define the propositional formulae*

$$\mathit{Init} \equiv \Big(\bigwedge_{i \in I^{(-1)}} \neg i\Big) \wedge T(W_{-1}, W_0) \qquad T_n \equiv \bigwedge_{0 \le i \le n-1} T(W_i, W_{i+1})$$

where $n \geq 0$ and $T(W_i, W_{i+1}) \equiv \psi\,[W_i/(I \cup C)][W_{i+1}/(I' \cup C')]$.

Given a ladder logic formula ψ, then the formula $Init \wedge T_n$ is "satisfied" exactly by all state sequences $s_0, s_1, \ldots, s_n$ of $A(\psi)$. More formally: Given a state sequence $s_0, s_1, \ldots, s_n$ we construct an valuation $\mu : W_{-1} \cup W_0 \cup \cdots \cup W_n \rightarrow \{1, 0\}$, where state s_j gives the interpretation of W_j for $0 \leq j \leq n$, i.e. $\mu(i^{(j)}) = s_j(i)$, $i \in I$, and $\mu(c^{(j)}) = s_j(c)$, $c \in C$; $\mu(i^{(-1)}) = 0$, $i \in I$, and $\mu(c^{(-1)})$ such that we reach s_0 via ψ. For this μ holds: $\mu \models Init \wedge T_n$. Conversely, given a μ with $\mu \models Init \wedge T_n$ one can decompose it to a state sequences $s_0, s_1, \ldots, s_n$ of $A(\psi)$.

With these notations in place we can define safety at a specific point in a sequence $W_0, W_1, W_2, \ldots$

Definition 6 (Safety at step n). *Let φ be a safety condition for a ladder logic formula ψ. We define the propositional formula*

$$\varphi_n \equiv \varphi\,[W_{n-1}/(I \cup C)][W_n/(I' \cup C')],$$

where $n > 0$.

6.2 Bounded model checking

Widely used within industrial applications [9, 4], bounded model checking restricts the search space by a bound which states how many transitions of the automaton should maximally be considered for the verification process. Using the formulae

$$\mathit{Initial} \equiv \mathit{Init} \wedge T(W_0, W_1) \Rightarrow \varphi_1 \text{ and } \mathit{Transition}_n \equiv T_n \Rightarrow \varphi_n,$$

the algorithm shown in Figure 4 performs a forwards iteration of the state space. Given a automaton $A(\psi)$ and safety condition φ, the algorithm will check: (1) that φ holds on all transitions leaving the initial states of the automaton, and that (2) φ holds for up to K transitions from an initial state of the automaton.

```
j ← 1
if ¬Initial is satisfiable return error trace
j ← j + 1
while j ≤ K do
        if ¬Transition_j is satisfiable return error trace
        j ← j + 1
return "K-Safe"
```

Fig. 4. K-step forwards iteration algorithm.

The above algorithm calls a SAT solver once in every iteration. In practice, the algorithm performs better when $l > 1$ calls, namely "$\neg \mathit{Transition}_j$ satisfiable", ..., "$\mathit{Transition}_{j+l}$ satisfiable", to the SAT solver are combined to *one* call, namely "$\neg(\mathit{Transition}_j \wedge \cdots \wedge \mathit{Transition}_{j+l})$ satisfiable", to the SAT solver.

Practical results from the pelican crossing example, show that verification times are less than one second[2]. With inductive verification, verification of the safety condition given in Section 5.1 fails for the induction step. With the proposed bounded model checking approach, we were able to show that this was in fact due to unreachable states. To show this, a bound size of six was required, along with meta-reasoning about the automaton, namely, that six steps cover all of its reachable states.

6.3 Unbounded model checking

Temporal induction [18] is a method that is based on strengthening the inductive approach as e.g., given by Kanso [15]. As the name suggests, the verification method still consists of two proof steps, namely a base case and an inductive step. These proof steps are however used differently: the (negation of the) base case is checked if it is satisfiable, the (negation of the) inductive step is checked if it is unsatisfiable. Our presentation follows [10].

We define properties of a state sequence encoded by $W_0, W_1, \ldots, W_n$:

$$LF_n \equiv T_n \wedge (\bigwedge_{0 \leq k < l \leq n} \neg(W_k \Leftrightarrow W_l)) \text{ and } safe_n = \bigwedge_{1 \leq j \leq n} \varphi_j.$$

where $(W_k \Leftrightarrow W_l) \equiv \bigwedge_{i \in I} i^{(k)} \Leftrightarrow i^{(l)} \wedge \bigwedge_{c \in C} c^{(k)} \Leftrightarrow c^{(l)}$; $k, l, n \geq 0$. LF_n describes the state sequences of length n of an automaton which are "loop free", i.e. the states appearing in the sequence are pairwise different. The formula $safe_n$ encodes that all transitions between two consecutive states are safe. Using these formulae, we can define base case and induction step of temporal induction:

$$Base_n \equiv \text{Init} \wedge T_n \Rightarrow \varphi_n \text{ and } Step_n \equiv T_{n+1} \wedge LF_{n+1} \wedge safe_n \Rightarrow \varphi_{n+1}.$$

Figure 5 gives the temporal induction algorithm, similar to [18, 9].

$n \leftarrow 0$
while true do
    if $\neg Base_n$ is satisfiable return trace
    if $\neg Step_n$ is unsatisfiable return "Safe"
    $n \leftarrow n + 1$

Fig. 5. Temporal induction algorithm.

Theorem 2. *For all ladder logic formulae and safety conditions, temporal induction terminates, is sound, and is complete.*

[2] All results presented in this paper are based on tests carried out using a 64-bit computer, with a 3GHz quad-core processor and 8 GBytes of memory.

Proof. (Only termination) Let ψ be a ladder logic formula. Let φ be a safety condition. Given that the automaton $A(\psi)$ is finite, we know that for some k all state sequences longer than k include a state twice. Thus, the formula $T_{k+1} \wedge LF_{k+1}$ is unsatisfiable. This implies that $Step_k \equiv T_{k+1} \wedge LF_{k+1} \wedge \varphi_k \Rightarrow safe_k$ is a tautology. Hence $\neg Step_k$ is unsatisfiable.

The temporal induction algorithm fully automatically verifies our pelican crossing example; no meta-reasoning was required. Once again, the verification time was less than one second.

7 Program slicing

The proposed approaches for the verification of ladder logic programs quickly give rise to large formulae to be verified. As the formula size increases, both the space and time requirements increase. This increase leads to a rather small bound[3] on the number of iterations of a ladder logic program we can verify in a feasible amount of time.

In our setting, the intuition behind slicing is that given a particular safety condition for verification, the variables that occur within the safety condition are only dependant on some part of the ladder logic program. Hence parts that have no effect on the safety condition can be removed.

7.1 Algorithm for slicing ladder logic

We define the dependence between rungs in a ladder logic formula.

Definition 7 (Dependency relation). *Let $\psi = [R_1, R_2, \ldots R_n]$ be a ladder logic formula for some $n \geq 0$. We define the relation dependant $\subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ between rungs of the ladder logic program, as the transitive closure of*

$$\{(i, j) \mid j < i \text{ and } c'_j \in vars(\psi_i)\}$$

where rung k has the form $R_k \equiv c'_k \Leftrightarrow \psi_k$ for $1 \leq k \leq n$.

Using this notion of dependence, we define the slice of a ladder logic formula w.r.t. a safety condition as:

Definition 8 (Slice). *Given a ladder logic formula $\psi = [R_1, R_2, \ldots R_n]$, and a safety condition φ. A slice ψ_φ of ψ is an order preserving selection of rungs such that the following two conditions hold:*

- *$R_j \in \psi_\varphi$ if $c_j \in vars(\varphi) \vee c'_j \in vars(\varphi)$.*
- *$R_j \in \psi_\varphi$ if $R_i \in \psi_\varphi$ and $(i, j) \in dependant$.*

Given a slice ψ_φ we define the sets

$$\hat{I} = vars(\psi_\varphi) \cap I \text{ and } \hat{C} = \{c \in C \mid c' \in vars(\psi_\varphi) \cap C'\}$$

of those input variables (state variables) that appear in the slice.

[3] I.e., with 599 variables, approximately 100 iterations were possible.

Note that this definition does not include a notion of minimality. Consequently, a ladder logic formula ψ is always a slice of itself. If the safety condition $\varphi \equiv True$, then for every ladder logic formula ψ we have that the empty programme $\psi_\varphi \equiv true$ is a slice. To ensure rung order is maintained, we compute a slice in a backward fashion. The algorithm we present is due to [11].

Step 1 – Extract variables from safety condition. Given a safety condition φ of the form described in Section 5.1, we extract its variables: $U = vars(\varphi)$.

Step 2 – Calculate dependant variables. Calculate all the variables of the ladder logic formula that effect the variables in U. This step is repeated for each rung until a fixed point within the variable set is reached. Figure 6 illustrates the code that could be used to perform this step.

$U_{n+1} \leftarrow U$
do
  $\overline{U} \leftarrow U$
  $U_{n+1} \leftarrow U$
  for $i = n$ down to 1 do
    if $c_i' \in U_{i+1}$ then $U_i \leftarrow U_{i+1} \cup vars(\psi_i)$ else $U_i \leftarrow U_{i+1}$
  $U \leftarrow U_1$
until $U \subseteq \overline{U}$
return $\overline{U}$

Fig. 6. Algorithm to compute step two.

Step 3 – Extract dependant rungs. Finally, using the variable set $\overline{U}$ computed in step two, we remove all rungs that do not effect the safety condition. To do this, we construct the set

$$index = \{i \in \{1, \ldots, n\} \mid c_i \in \overline{U} \text{ or } c_i' \in \overline{U}\}.$$

Now, we remove from the original program all rungs R_i whose indicies do not appear in $index$. The result ψ_φ is the sliced version of program ψ.

Figure 7 illustrates the effect of slicing the ladder logic formula of Figure 2 w.r.t. the safety condition presented in Section 5.1: The safety condition has four variables, six out of the original eleven rungs remain.

$$\begin{aligned}
&[\ crossing' \iff (req \wedge \neg crossing),\\
&\ req' \iff (pressed \wedge \neg req),\\
&\ tlag' \iff ((\neg crossing') \wedge (\neg pressed \vee req')),\\
&\ tlbg' \iff ((\neg crossing') \wedge (\neg pressed \vee req')),\\
&\ tlar' \iff crossing',\\
&\ tlbr' \iff crossing']
\end{aligned}$$

Fig. 7. A Sliced version of our pelican crossing ladder logic formulae.

7.2 Correctness of slicing

Given that slicing changes the ladder logic formulae under consideration, we need to ensure that the validity of safety conditions is still upheld.

Throughout this Section we assume that ψ_φ is the slice of a ladder logic formula $\psi = [R_1, R_2, \ldots R_n]$ w.r.t. a safety condition φ, where $\hat{I}$ is the set of inputs of ψ which appear in ψ_φ and $\hat{C}$ is the set of state variables of ψ required by ψ_φ – see Definition 8.

In order to compare the two automata $A(\psi)$ and $A(\psi_\varphi)$ we first need to relate their states. $A(\psi)$ has maps $\mu : (I \cup C) \to \{0,1\}$ as its states, while the states of $A(\psi_\varphi)$ take the form of maps $\nu : (\hat{I} \cup \hat{C}) \to \{0,1\}$. To this end, we define two functions: $\_\_|_{\hat{I}\cup\hat{C}}$ maps states from $A(\psi)$ to states from $A(\psi_\varphi)$, $\_\_ :: f$ maps a state from $A(\psi_\varphi)$ to a state of $A(\psi)$, where f is a valuation that describes how we interpret the variables in $(I \cup C) - (\hat{I} \cup \hat{C})$.

Definition 9 (Reducing/Extending a valuation).

1. *Let μ be a state of $A(\psi)$. Its reduction $\mu|_{\hat{I}\cup\hat{C}} : \hat{I} \cup \hat{C} \to \{0,1\}$ w.r.t. $\hat{I} \cup \hat{C}$ is defined as $\mu|_{\hat{I}\cup\hat{C}}(x) = \mu(x)$ for all $x \in \hat{I} \cup \hat{C}$.*
2. *Let ν be a state of $A(\psi_\varphi)$. Let $f : (I \cup C) - (\hat{I} \cup \hat{C}) \to \{0,1\}$ be an evaluation. We define the extension of ν by f as $(\nu :: f) : C \cup I \to \{0,1\}$ where*

$$(\nu :: f)(x) = \begin{cases} \nu(x) & \text{if } x \in \hat{I} \cup \hat{C} \\ f(x) & \text{otherwise} \end{cases}$$

for all $x \in C \cup I$.

Remark 2. We also apply reduction and extension to paired valuations. That is, $(\mu \fatsemi \mu')|_{\hat{I}\cup\hat{C}} = (\mu|_{\hat{I}\cup\hat{C}}) \fatsemi (\mu'|_{\hat{I}\cup\hat{C}})$ is the paired evaluation obtained from individually reducing μ and μ'. $\nu :: f \fatsemi \nu' :: f' = (\nu :: f) \fatsemi (\nu' :: f')$ is the evaluation obtained by individually extending ν by f and ν' by f' and then pairing the results.

We now study how to relate transitions of $A(\psi)$ to transitions of $A(\psi_\varphi)$: A step in $A(\psi)$ corresponds to a step in $A(\psi_\varphi)$; consequently, reachability in $A(\psi)$ implies reachability in $A(\psi_\varphi)$.

Lemma 1 ($A(\psi)$ transitions correspond to $A(\psi_\varphi)$ transitions). *Let μ and μ' be states of $A(\psi)$.*

1. *$\mu \fatsemi \mu' \models \psi \Rightarrow \mu \fatsemi \mu'|_{\hat{I}\cup\hat{C}} \models \psi_{P\varphi}$*
2. *If $\mu \fatsemi \mu'$ is reachable with respect to $A(\psi)$ then $\mu \fatsemi \mu'|_{\hat{I}\cup\hat{C}}$ is reachable with respect to $A(\psi_{P\varphi})$.*

Corresponding results hold for the reverse direction:

Lemma 2 ($A(\psi_\varphi)$ transitions can be extended to $A(\psi)$ transitions). *Let ν and ν' be states of $A(\psi_\varphi)$.*

1. *Let* $\nu \wr \nu' \models \psi_\varphi$. *Then for all* f *there exists a* f' *such that* $\nu :: f \wr \mu' :: f' \models \psi$.
2. *Let* $\nu \wr \nu'$ *be reachable with respect to* $A(\psi_\varphi)$. *Then there exist* f, f' *such that* $\nu :: f \wr \mu' :: f'$ *is reachable with respect to* $A(\psi)$.

Using these lemmas we can prove that slicing is correct:

Theorem 3. *Let* φ *be a safety condition over a ladder logic formula* ψ. *Then*

$$A(\psi) \models \varphi \iff A(\psi_\varphi) \models \varphi.$$

8 Application and results

We summarise some results that have been obtained via a verification tool based on the discussed methods. A detailed discussion of the implementation of the tool, and the results are available in [13]. In total, two railway interlocking ladder logic programs were verified, one containing 331 rungs with 599 variables, and the other 238 rungs with 361 variables in total.

Overall, the results we have gained have been positive. For every safety condition we have verified, the tool has either given a successful verification, or a counter example trace. All results have been obtained within the region of seconds.

8.1 Results of bounded model checking

The main success of the forward iteration approach proved to be the generation of counter example traces. In all the verification results where inductive verification via Kanso's method [15] gave a counter example, the forward iteration approach was successfully able to construct a counter example trace. This demonstrates the need of tool supported verification: It turned out that the claimed to be unreachable states – see the Introduction – actually are reachable. However, they do not effect the safety of the system as they can be excluded thanks to invariants. Such invariants have not been considered in our automaton model, as in industry they are soft constraints known only to engineers. Such constraints, however, are not part of the documentation for the interlocking control programs.

Results obtained show that forward iteration was possible up to two thousand iterations before memory issues occurred. With the application of our slicing algorithm, the number of iterations possible increased to twenty thousand. This is a large number of iterations, however, it remains unknown how many iterations would be required to verify all reachable states.

8.2 Results of temporal induction

The results obtained from the temporal induction approach are as expected. Whenever inductive verification via Kanso's method [15] succeeded, i.e., the safety property held, the safety property was also provable via temporal induction. Also, whenever a counter example was generated using the forward

iterations technique, a counter example would be generated by temporal induction. These two results show that temporal induction works correctly. The full power of temporal induction, however, is demonstrated by our Pelican crossing example: only temporal induction is capable of verifying it fully automatically.

8.3 Results of slicing

All results obtained show that applying slicing to the formulae to be verified resulted in large efficiency gains. Some analysis of the application of the slicing algorithm have shown that the following reductions were possible:

- For interlocking one, the number of rungs contained in the ladder logic formula, was reduced, on average from 331 rungs to around 60 rungs.
- For interlocking two, the number of rungs contained in the ladder logic formula, was reduced, on average from 238 rungs to around 25 rungs.

Obviously, the resultant formula size is dependant on the safety condition being verified. Hence it would be interesting to see the effect slicing has on more complicated, larger interlockings.

9 Conclusion

We have completed a feasibility study into various techniques for SAT-based model checking of Westrace interlockings. We have provided a modelling process for Westrace interlockings via propositional logic and given an automaton theoretic semantics for this propositional model. We have studied in some depth, the verification processes of bounded model checking and unbounded model checking via temporal induction. As a natural continuation from this, we have reviewed how a slicing algorithm can be applied to reduce the complexity of the verification problem, showing the correctness of its application. The overall outcome being the development of a verification tool, with varied verification techniques on offer. This tool has been applied to verify real world interlockings, with the main results being:

- The approaches we propose work. That is, an interlocking can successfully be verified with respect to some safety condition. The result being either that the interlocking is safe, or that a counter example trace is generated.
- The approaches we propose scale up to real world systems.
- SAT-based verification is a successful method of verifying large systems.

Future work will include the removal of functional dependencies [14] and the verification of further interlockings.

Acknowledgements

Thanks to Fredrik Nordvall Forsberg and Liam O'Reilly for giving valuable feedback towards our work, and to Erwin R. Catesbeiana (Jr) for keeping us on track throughout.

References

1. Programmable Controllers - Part 3: Programming languages, 2003. IEC Standard 61131-3.
2. Uppaal tool, Webpage, last accessed in October 2009. `http://www.uppaal.com/`.
3. Invensys Rail, Webpage, last accessed October 2009. `http://www.wrsl.com/assets/files/Interlocking/westrace/WESTRACE%20Intorduction.pdf`.
4. N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In D. Borrione and W. J. Paul, editors, *CHARME*. Springer, 2005.
5. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *TACAS '99*. Springer-Verlag, 1999.
6. A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.
7. D. Bjoerner. Towards a domain model of transportation. In *Domain Engineering – Technology Management, Research and Engineering*. JAIST Press, 2009.
8. J. Boulanger and M. Gallardo. Validation and verification of METEOR safety software. In *Advances in Transport Vol 7*. WIT Press, 2000.
9. K. Claessen, N. Een, M. Sheeran, and N. Sörensson. SAT-solving in practice. In B. Lennartson, M. Fabian, K. Akesson, A. Giua, and R. Kumar, editors, *Proceedings of Workshop on Discrete Event Systems (WODES)*. IEEE, May 2008.
10. N. Een and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. BMC'2003.
11. W. Fokkink and P. Hollingshead. Verification of interlockings: from control tables to ladder logic diagrams. In J. Groote, S. Luttik, and J. van Wamel, editors, *FMICS'98*. CWI, 1998.
12. R. Jacquart, editor. *IFIP 18th World Computer Congress, Topical Sessions*, chapter TRain: The Railway Domain - A Grand Challenge. Kluwer, 2004.
13. P. James. SAT-based Model Checking and its applications to Train Control Software. Master's thesis, Swansea University, 2009.
14. J.-H. R. Jiang and R. K. Brayton. Functional dependency for verification reduction. In R. Alur and D. A. Peled, editors, *CAV*. Springer, 2004.
15. K. Kanso. Formal verification of ladder logic. Master's thesis, Swansea University, 2008.
16. K. Kanso, F. Moller, and A. Setzer. Verification of safety properties in railway interlocking systems defined with ladder logic. In M. Calder and A. Miller, editors, *AVOCS08*, Glasgow 2008.
17. O. Kullmann. The OKlibrary: A generative research platform for (generalised) SAT solving. Technical Report CSR 1-2008, Swansea University, 2008.
18. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT-solver. *Lecture Notes in Computer Science 1954*, 2000.
19. K. Winter. Model checking railway interlocking systems. *Australian Computer Science Communications*, 24(1), 2002.
20. B. Zoubek, J.-M. Roussel, and M. Kwiatowska. Towards automatic verification of ladder logic programs. In *Proceedings of IMACS-IEEE (CESA'03)*, 2003.

Towards Formal Algebraic Modeling and Analysis of Communication Spaces*

Tony Modica

modica@cs.tu-berlin.de
Integrated Graduate Program Human-Centric Communication
Technische Universität Berlin

Abstract. We subsume Communication Spaces (CS) as communication-based systems taking into account the central notions of interpretation of content in contexts, communication roles, and allowing for human-centric demands, e.g. adaption to environment and preferences. Since most of the well-known formal modeling approaches are adequate only for specific aspects or limited views of systems considered as CS, in this article a new formal approach is advocated. This approach is an integration and extension of the well-established modeling techniques of algebraic high-level Petri nets and rule-based graph transformation, intended to cover the main aspects of CS and to analyze and verify properties specific to them. We demonstrate the new approach of Algebraic Higher-Order Net with Individual Tokens (AHOI nets) on an example modeling of the widely known Internet telephone software Skype. This allows us to discuss the advantages of AHOI nets w.r.t. needs of a basic modeling of CS.

1 The Challenges of Communication Spaces

The notion of *Communication Spaces* (CS) is not intended to be fixed formally. Moreover, it is meant to serve as a characterizing concept of communication systems featuring specific aspects, e.g.:

- *Contextuality* of contents that is transmitted (via *channels*) by communicating entities (*actors*).
- *Dynamics* in the system structure, so that actors may move in CS, even join or leave several different CS.
- *Preferences*, *access rights*, and *roles* are to be respected.

Typical examples that can be considered from the CS viewpoint are Internet-based applications like Skype, Facebook, or SecondLife; and also Mobile Ad-hoc networks or SmartHomes, in which appliances are connected intelligently

* This work has been supported by the Integrated Graduate Program on Human-Centric Communication at TU Berlin (http://www.h-c3.org/ra_en.html#RAE) and by the research project forMAlNET (http://tfs.cs.tu-berlin.de/formalnet) of the German Research Council

to offer increased comfort to their inhabitants. It is desirable to have a formal modeling technique for CS, so that we can specify the features of such systems unambiguously and are able to simulate, test, and analyze/verify[1] them, using the formal semantics of the modeling technique. We have observed that most of the well-known modeling techniques like UML and actor systems [1] or formal specification techniques like process algebras [2], low-level and high-level Petri nets [3,4], algebraic specification [5] and graph transformation [6], and different kinds of logic are only adequate to model and/or analyze specific aspects of CS. Plain Petri nets for example have a static structure. Graph transformation systems in contrast are dynamic in their structure but lack a description of system behaviour. Of course, appropriate graph transformation systems may also be used to simulate e.g. the behaviour of Petri nets but it seems advisable to distinguish system behaviour from reconfiguration and to possibly use standard results for analysis. Another thing to mention is that we believe that visual diagrammatic models as Petri nets and graphs can have advantages for system modeling w.r.t readability and understandability, though there is no standard measure for these properties.

As a consequence of the above, we advocate a new integrated formal modeling technique for CS. In this paper, however, we only give an informal introduction to our new visual modeling technique and demonstrate how it can be used to model a typical example of a CS: the Internet telephone software Skype.

2 Modeling of CS with a new Kind of Petri Nets

In this section, we formulate some main requirements for modeling of CS and give an informal overview to a modeling approach, called *AHOI Petri nets*.

2.1 Requirements for Modeling of CS

An adequate formal modeling approach would have to cover at least three main aspects of CS:

1. Data and content in the CS and the knowledge of actors (their *content spaces*).
2. Structure of the CS, which actors are connected to each other (in short, the *topology*). In particular, the structure should be dynamic to allow actors to enter and leave.
3. *Interactions* in and between different CS, transmission of data

According to our experience, no single classic modeling approach is powerful enough to achieve this. For this reason, we propose a new integrating approach: *reconfigurable **A**lgebraic **H**igher-**O**rder Petri nets with **I**ndividual tokens* (AHOI nets).

[1] General interesting properties are realted to consistency, safety and security requirements, liveness, termination etc.

Definition 1 (Algebraic Higher-Order Nets with Individual Tokens). *An Algebraic Higher-Order Net with Individual Tokens (AHOI net) is a tuple* (AN, I, m) *where*

- *AN is an Algebraic High-Level (AHL) Net as in [7] whose algebra features a suitable token sort for some kind of Petri net and operations for calculating activated transitions and the results of firing steps. The eponymous example with low-level place/transition nets as tokens are the Algebraic Higher-Order nets introduced in [8][2].*
- *I is the (possibly infinite) set of individual tokens of the AHOI net.*
- $m : I \to (A_{AN} \times P_{AN})$ *is the marking function, assigning each individual token to a pair* (a, p)*, representing an algebraic value* a *on place* p *in* AN*.*

A reconfigurable AHOI net is an AHOI net with a set of transformation rules. The main advantage of Petri nets with individual tokens to Petri systems following the collective token approach (where marking are elements of the commutative monoid $(A_{AN} \times P_{AN})^{\oplus}$) is that we can formulate transformation rules in the sense of the double-pushout (DPO) approach in [6], so that these rules also can change markings. In regular AHL nets, i.e. with collective token markings, changing of markings via DPO transformation is not possible due to technical restrictions. The firing behaviour of an AHOI net (AN, I, m) with finite I is equivalent to the behaviour of AN with the marking $\sum_{i \in I} m(i)$.

As we will see in the next section's example, AHOI nets are powerful enough to cover the main aspects of CS by integrating Petri nets (topology), abstract data types (content spaces), and net transformation (interaction and dynamics). However, to be able to follow this example, we first give an informal introduction to AHOI nets in a simplified notation, while a formal theory of AHL nets with individual tokens is under development in a technical report to appear.

2.2 AHOI Nets in a Nutshell

Instead of reviewing Algebraic High-Level nets we will go through a quick tutorial of AHOI nets, for which we avoid the formal notation and consider a simplified visual representation: Fig. 1 shows an AHOI net component, with rectangles as *transitions* and ovals as *typed places* that contain *tokens*. Arcs inscripted with terms connect places and transitions to form a bipartite graph. For example, there are the places *User* of type *SkypeName*, *Template* of type *DataUnit*, and *State* of type *State*, containing e.g. the tokens *Alice* and *Offline*, which represent values of the correspondent data types. Especially, we have a token *DU(Alice)* being itself a Petri net[3], which we consider as *higher-order* tokens (or *object nets*, to distinguish them from the containing *system net*). These three places constitute the *predomain* of transition *activate* because of the arcs pointing to

[2] The idea of nets in nets stems from [9] as an approach to represent mobile agents in a Petri net

[3] This net is represented as a value of an appropriate algebraic type for Petri nets.

the transitions from these places. Similarly, *User*, *Online*, *Ready2Talk*, and *Template* are the *postdomain* of *activate*, because of the arcs pointing to them from *activate*.

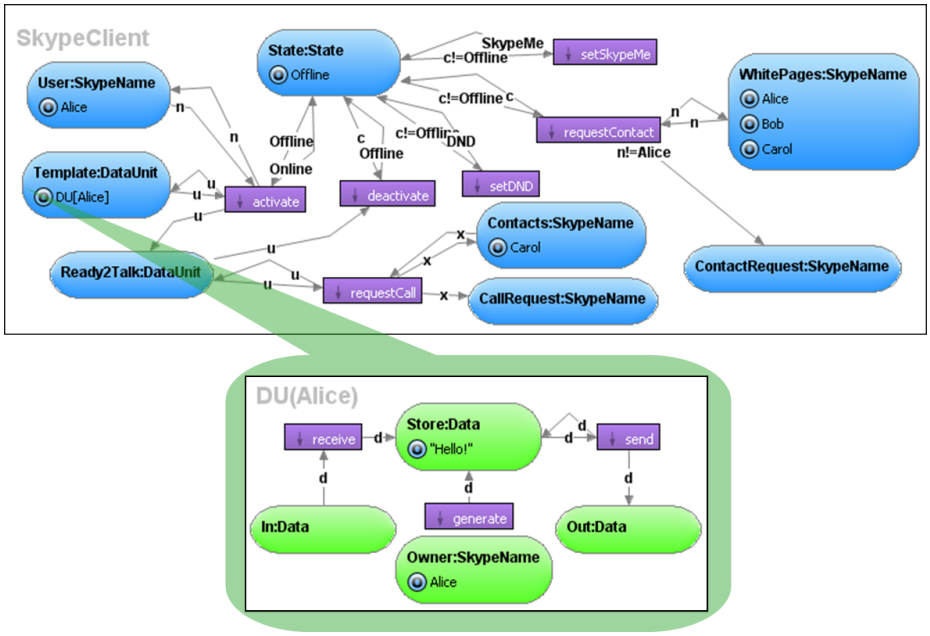


Fig. 1. AHOI net component for a Skype client

We call a transition *enabled* if on each of its predomain places we can find a token so that the set of selected tokens is consistent with the arc inscriptions. Basically, a token selection is consistent if the tokens correspond with the definite arc terms and we can find a consistent assignment for the arc variables to the selected tokens[4]. In Fig. 1 for example, activate is enabled because we can assign *u=DU(Alice)*, *n=Alice*, and the token *Offline* corresponds directly to the arc's term. If a transition is enabled, we can *fire* it, which means that the predomain tokens are removed and for each term on the postdomain arcs, a token according to the variable assignment and arc inscriptions is added to the corresponding place.

[4] Actually, in general AHOI nets arcs can have arbitrary terms with variables, e.g. $op(y, z, \ldots)$, and transitions additionally may have firing conditions like $op_1(x) = op_2(y, z, \ldots)$ according to the signature of the system net's algebra. But for the example, we only consider definite values and variables as arc inscriptions, as well as the simple condition pattern *v!=Val*, denoting that additionally the token value *Val* can not be assigned to variable *v*.

Note that e.g. place *User* is in the predomain and also in the postdomain of *activate* with the same arc inscription on the in- and outgoing arc, so its marking will not change on firing *activate*. Due to the arc to *Ready2Talk* inscripted with the same variable u as the arcs from and back to *Template*, firing *activate* copies the token assigned to u to both places. In short, if we fire *activate*, the result will be that *DU(Alice)* will have been copied to *Ready2Talk* and the token *Offline* on *State* will be replaced by token *Online*.

To reconfigure AHOI nets we use the rule-based net transformation approach in [6,10]. Transformation rules are spans of morphisms $L \leftarrow K \rightarrow R$ with a left-hand side L, a right-hand side R, and an interface K being the intersection of L and R, all these being AHOI nets. In Fig. 2, we consider an example AHOI net rule $p : L \leftarrow K \rightarrow R$ to demonstrate transformation: In general, to apply p to a net N we need to find an occurrence of L in N, specified by an injective AHOI net morphism $o : L \rightarrow N$. To get the context C, we remove everything that is matched by parts of L and is not preserved in K and finally add in the result net N' everything found in R that is not already present in the interface K. Altogether, Fig. 2 shows a net transformation via rule $p : L \leftarrow K \rightarrow R$, where N can be considered as gluing of L and C along K and N' as gluing of R and C along K[5]. Note that also the markings of places *p1* and *p3* are changed by this rule application.

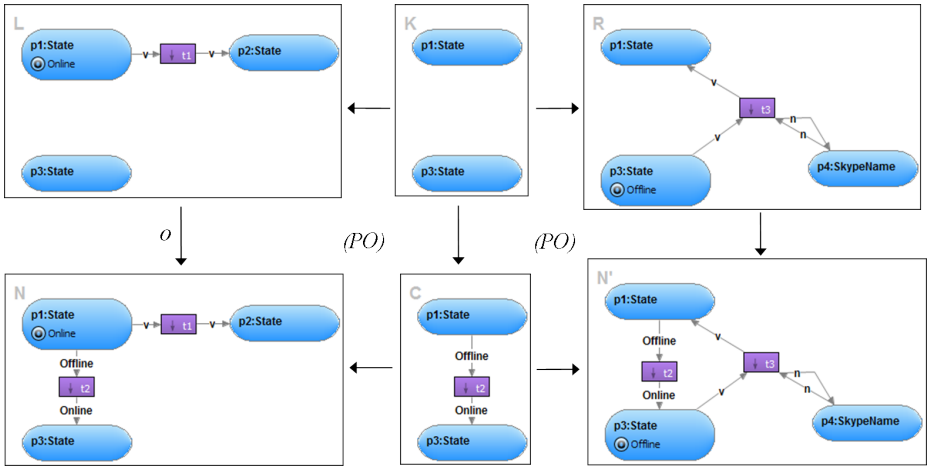


Fig. 2. Example AHOI net transformation

AHOI net morphisms (e.g. the match) must fulfill some structural conditions to ensure that rule applications yield valid AHOI nets [11]. Because L and R

[5] The resulting squares in Fig. 2 are pushouts in the category of AHOI nets, i.e. N is the gluing of L and C, where the interface K specifies items to be identified; analogously for N'.

are usually sufficient to understand which parts a rule deletes and creates, we mostly omit K and denote a rule as $L \to R$ in a compact notation.

3 Modeling Skype with AHOI Nets

Skype[6] is a widely used program for Internet telephony, offering easy to use (synchronized) data exchange and conferences. With its contact and privacy management, users can decide who and how other users can contact them. We discuss how Skype, as a typical representative offering many CS-relevant features, can be modeled with AHOI nets. First, we have to make some general decisions about what aspects we focus on and how to represent these.

Skype is not open source, there is no (publicly available) formal model, especially no one according to the CS viewpoint, and Skype uses proprietary network protocols. Therefore, we limit ourselves to modeling observable behavior only, i.e. to activities users can perform in their Skype client software and the direct effects of these activities caused in the Skype system. Our example follows these guidelines:

- A single AHOI net models the whole Skype system. Each user, resp. his client instance, is represented by an (initally discrete) component of this system net.
- We strictly distinguish user-triggered client behavior and system reactions. User actions are modeled by transitions in the corresponding client component; a user can act if at least one of its client's transitions is enabled. In contrast, global system actions are modeled by rules that reconfigure the system. To be more specific: An action in a client can either alter the client's configuration directly (like (de)activating the client, modifying privacy settings etc.) or represent a request to the Skype system to perform a global task (like establishing connections or transmitting data) that may extend/restrict possible actions of the client. System operations executing such requests are realized by rule applications, which possibly create or remove transitions of a client net component.
- To keep the intuitive visual representation of AHOI nets comprehensible, we assume the system to apply cleaning-up rules on temporary net structures after the activity that they were created for has been completed. Moreover, when simulating a system, the modeler should be able to grasp the system's state very quickly.

3.1 Skype User Clients as AHOI Components

Fig. 1 shows the AHOI component for the Skype client of a user "Alice". Each client component has the following basic structure:

[6] Skype is free to use and freely available at `http://www.skype.com`.

- One place typed by *State*, which is also named *State*, represents the current state of a Skype client. The data type *State* consists of the values *Offline*, *Online*, *SkypeMe*, and *DND*. The example client in Fig.1 is in state *Offline*.
- *SkypeName* places carry identities of Skype users, e.g. the token *Alice* on the place *User* indicates that Alice is the client's owner and the tokens on *Contacts* represent two Skype users she has in her contact list[7]. The place *WhitePages* is shared between all user clients and carries tokens corresponding to the tokens on the *User* places of all existing clients. A *SkypeName* token on place *CallRequest* would announce a request to the system for connection to the correspondent client, and similarly on place *ContactRequest* for exchange of contact data.
- *DataUnit* higher-order tokens, as shown in the left of Fig. 1, are a kind of agents that allow their owner client to send data to and receive data from other clients. The owner is indicated by the *SkypeName* token on the unit's place *Owner* and we denote a unit with owner X as *DU(X)*. The type *Data* may represent audible, textual, graphical etc. data, which a unit can generate, send, and receive by firing its transitions.
- If there is a token on the client's place *Ready2Talk* the client is supposed neither to be offline nor to participate in another call/conference, hence to be able to accept incoming calls.

An example firing step for activating the client has been discussed in Sect. 2.2. The remaining possible firing steps in Fig. 1 are changing the client's state to DoNotDisturb or SkypeMe by firing the corresponding transitions, deactivating the client by firing *deactivate*, which would delete the *DataUnit* token from *Ready2Talk*, or announcing requests for a either a connection to another client by firing *requestCall* or for contact exchange by firing *requestContact*. In the following section, we will see how the system reacts on requests by reconfiguring clients to allow more activities.

3.2 Request for Contact

In the following we suppose a minimal example of a system net that contains two client net components like the one shown in Fig. 1[8]. One has, as depicted, the tokens *Alice* on its *User* place, *Carol* on its *Contacts* place, and the object net token *DU[Alice]* on its *Template* place. The second is supposed to belong to a user named Bob, hence having the tokens *Bob* on its *User* place, *DU[Bob]* on its *Template* place and an empty *Contacts* place. Both clients are considered to be online, having the corresponding token on their *State* place, respectively.

[7] The data type *SkypeName* is some appropriate type for distinguishing identities, e.g. unique strings or integers.

[8] Note that these client components can be created in a system net (and be deleted) with reconfiguration rules dynamically! This realizes registration and resigning of Skype users. Basically, this rule consists of an empty left-hand side and the net in Fig. 1 as the right-hand side.

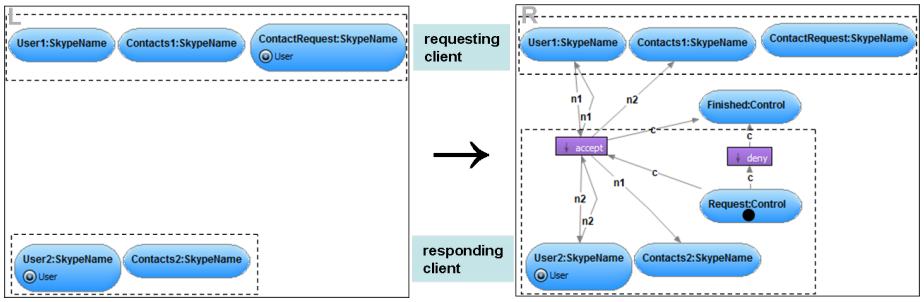


Fig. 3. Rule *CreateContactExchange* creating structure for contact exchange

Alice now wants to talk to Bob, which she would realize by firing *requestCall*, so that a token *Bob* is put on her *CallRequest* place. But she does not have his *SkypeName* token on her *Contacts* place, yet. So, before calling him she has to ask for his permission to add his contact to her contact list, represented by her *Contacts* place. This is according to the default procedure in Skype to follow.

To accomplish an exchange of user contacts, Alice fires her *requestContact* transition, so that the token *Bob* (assigned to the variable n in this firing step) is copied from the *WhitePages* place to Alice's *ContactRequest* place.

Remember, we want to separate strictly user behavior in the client from reactions by the system, following the modeling principles we stated at the beginning of this section. So, we let the Skype system react the Alice's request by applying the transformation rule *CreateContactExchange* shown in Fig. 3, extending the possible behavior of Bob's client.

In the left-hand side L of this rule we have three places that should be matched on the corresponding places of the requesting client and two places of the responding client[9]. Further, to be applicable, this rule needs a token value that can be assigned to the token variable *User*, one on *ContactRequest* and *User2* each. In our example scenario this would be the two tokens *Bob* indicating the user of Bob's client component and the request Alice just has announced before.

The effect of applying this rule is the creation of the structure in the right-hand side R and removing the request token on the place matched by *ContactRequest*. In the manipulated system net, the two clients are now connected with this structure and we interpret the newly created transitions as additional behavior for Bob's client. E.g., Bob can fire the transition *deny*, which moves a simple control token (assigned to variable c) to the *Finished* place and concludes the contact exchange request without further effect. Alternatively, he may fire *accept* which, besides moving the control token, will copy a token *Alice* to the Bob's *Contacts* place and a token *Bob* to Alice's *Contacts* place. This fol-

[9] The dashed frames and the boxes describing the roles in the middle of the rule are just a hint to understand to which component the matched places should be connected. They are actually not part of this or the following rules' syntax.

lows from having applied the rule matching *User1* place to Alice's *User* place, which surely carries a token *Alice* that, on firing *accept*, is assigned to variable *n1* and hence copied to Bob's *Contacts* place. The assignment of the token *Bob* (matched by the rule's token variable *User* before) to variable *n2* and copying it to Alice's *Contacts* place happens analogously.

In short, after Bob has fired *accept*, he now effectively has Alice on his contact list and vice versa and Alice is finally able to call him.

3.3 Cleaning the Model by Removing Dispensable Structures

After a request for contact exchange has been accepted or denied the additional structure created can be removed because none of the new transitions can fire any more, due to the control token being moved to *Finished*. We just can remove this now useless structure by the reversed creation rule, i.e. a rule that has basically *CreateContactExchange's R* as left-hand side and *L* as right-hand side. We don't carry this out in more detail[10] in this article, it just should demonstrate how reversed rules can be used in principle to keep the overall model lean and clear without confusing left-overs.

3.4 Creating Conferences

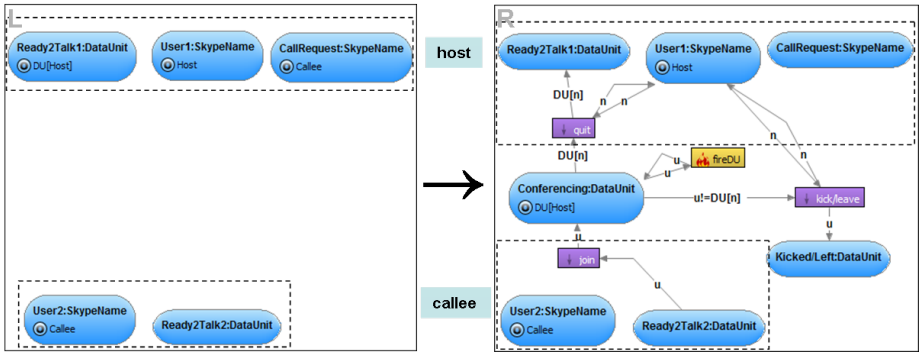


Fig. 4. Rule *CreateConference* creating a conference structure

Now, Alice wants to invite Bob to a direct call[11] and fires *requestCall* in her client component so that the token *Bob* that has been created before in

[10] Of course, the deleting rule is not supposed to delete a control token on the *Request* place and to create a *User* token as would do the simply reversed rule, but rather just to delete a control token on *Finished*.

[11] In Skype, you may invite additional contacts to a running conversation, so we consider a direct call just as an (initial) conference with two participants.

the contact exchange is copied to her *CallRequest* place. To allow the system to react to the request, we formulate the rule *CreateConference* depicted in Fig. 4. Similar to the previous rule, the four upper framed places in the left-hand side L should match the corresponding places of the conference host component, whereas the lower ones belong to the called client component. When applicated to our running example, the rule creates the conferencing structure in R, moves Alice's *DataUnit* token *DU[Alice]* from the *Ready2Talk* to the new *Conferencing* place, and deletes the request token *Bob* on *CallRequest*.

Being the host, Alice is attending the conference immediately after rule application; she is unavailable to other calls while her conference is running. Her only option is to *quit* (by firing the appropriate transition) and terminate the whole conference[12]. Bob may *join*, which would move his *DataUnit* token to *Conferencing* as well.

The conference is established now and we discuss the transitions *fireDU* and *kick/leave* in the following subsections.

3.5 Transmitting Data

We assume that Bob has joined the conference, so that his *DataUnit* token is now on the *Conferencing* place that just has been created by the rule *CreateConference*. Alice fired *send* in her *DataUnit* object net *DU[Alice]*[13], which copied the token *"Hello!"* from her unit's *Storage* to its *Out* place (cf. the left of Fig.1). We interpret a token on a *DataUnit's Out* place as a request to distribute the token value to all other *DataUnits* in the same conference.

The (vertically depicted) rule *Transmit* in Fig. 5 is a schema[14] whose instances each match a *DataUnit* token *sender* and a fixed number of receiver units to realize multicasting of data on a conference place; in our example we consider just Bob as a single receiver. In the right-hand side R, the matched tokens are replaced with algebraically calculated object nets, for which we use the following operations that we provide in the AHOI net's algebra:

- $out : DataUnit \rightarrow Data$ yields the token value on the *Out* place of the *DataUnit* passed as argument of this operation.
- $send : DataUnit \times Data \rightarrow DataUnit$ returns the passed *DataUnit* but removes the value of the *Data* argument from the unit's *Out* place.

[12] We assume the transition *quit* to have a firing condition, so that only *DU[Alice]* can be assigned to u. This is the reason why we need *quit* to be connected to Alice's *User* place; it needs to access her *SkypeName* token value.

[13] To fire object net transitions, we use the *fireDU* transition that has been created with the conference structure. It takes an object net token assigned to variable u, calculates algebraically the net that results after firing an enabled transition inside the object net u represents, and returns the fired net as a new object net token back to the conference place. For this we make use of the assumed operations on the Petri net token sort of AHOI nets. You may look at [8] for details of this construction.

[14] For now, we assume that we have a rule for each possible conference size, i.e. the number of participants. Recently, we proposed the more flexible and advanced approach of *Amalgamated Rules* for multicasting in [12].

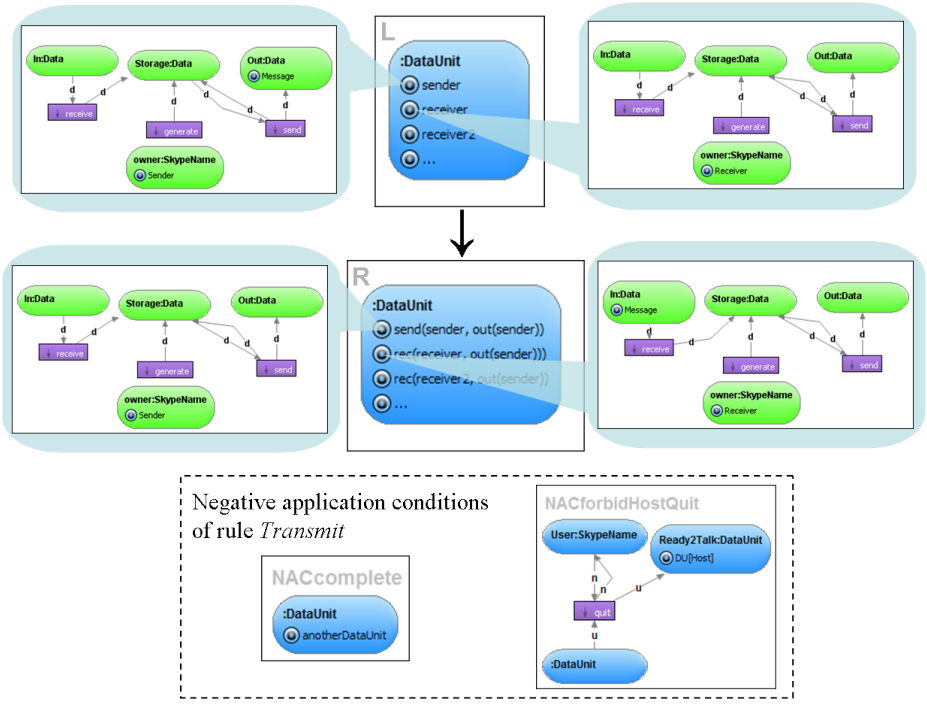


Fig. 5. Rule schema *Transmit* distributing data among *DataUnit* tokens

– $rec : DataUnit \times Data \rightarrow DataUnit$ is similar to operation *send* but returns the passed *DataUnit* where the *Data* argument value has been added to the unit's *In* place.

Now we can understand the terms in the right-hand side of *Transmit*: $send(sender, out(sender))$ is basically the *DataUnit sender* where the token has been removed from its *Out* place. Similarly, each term $rec(receiverX, out(sender))$ is the corresponding receiver *DataUnit* to which the *Data* token from the *sender* is added to. The green object nets besides the rule illustrate this. After applying this rule to our example, Bob can fire *receive* in his *DataUnit* to gather the transported *Data* from his input place *In* and the transmission has been completed.

To avoid incomplete transmissions (e.g. by applying a *Transmit* rule instance with just one receiver on a conference with three participants, hence with two receivers) and transmissions after the host has quit (which should immediately terminate the conference) we introduce *negative application conditions* (NACs) for rule *Transmit*. A rule is not applicable if there exists a valid occurrence for at least one of its NACs. With *NACcomplete* we ensure that the rule can be applied only if the left-hand side *L* matches all tokens on *Conferencing* so that there is not another unmatched *DataUnit* token left, allowing only complete transmissions to

all participants. *NACforbidHostQuit* prohibits a rule to be applied after the host has quit the conference (when his *DataUnit* can be located on his *Ready2Talk* place).

3.6 Joining and leaving Conferences

In Skype, the conference host may invite more participants to a running conference. In our example, Alice can fire another *callRequest* while she hosts a conference, so that the rule *InviteParticipant* in Fig. 6 can be applied. It simply connects another client to the conference like the intial rule for creating conferences before. Note that the invited participant does not necessarily have to be ready to talk to get invited, the rule rather ensures that Alice as the host has not quit the conference by requiring her *DataUnit* on the *Conferencing* place.

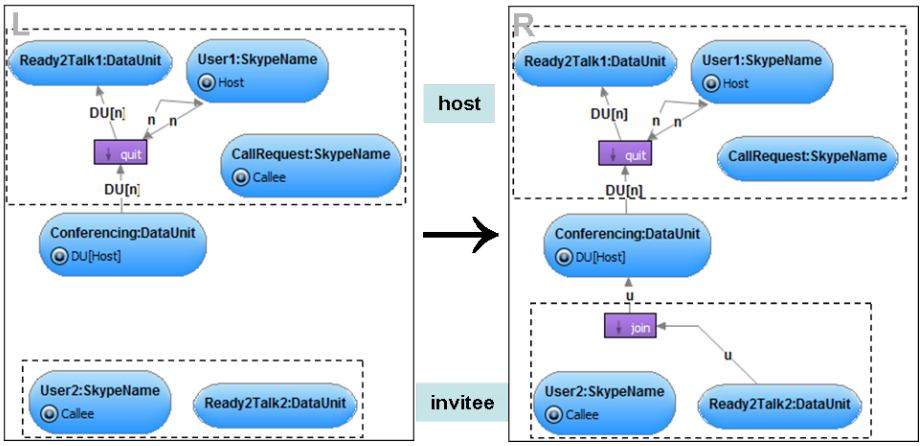


Fig. 6. Rule *InviteParticipant* connecting invited participants to conference

Every participant can leave the conference by firing *kick/leave*, but this can also be used by Alice to exclude participants from the conference. Note that Alice can only quit the conference via her own *quit* transition because the conference is considered to be finished when she does this. Other rules like *Transmit* respect that and do not allow to continue the conference so that the participants only option is to leave in this case. Again, a user firing *kick/leave* just announces a request that the system answers with application of rule *KickParticipant* in Fig. 7, which disconnects the client from the conference and moves its *DataUnit* token back to its *Ready2Talk* place.

It should be imaginable that an appropriate rule can be formulated to remove a conference whose participants all have left and whose host has quit, possibly with some NACs. This concludes the example scenario in a state where Alice's and Bob's clients have exchanged contacts and data and are now again unconnected client components.

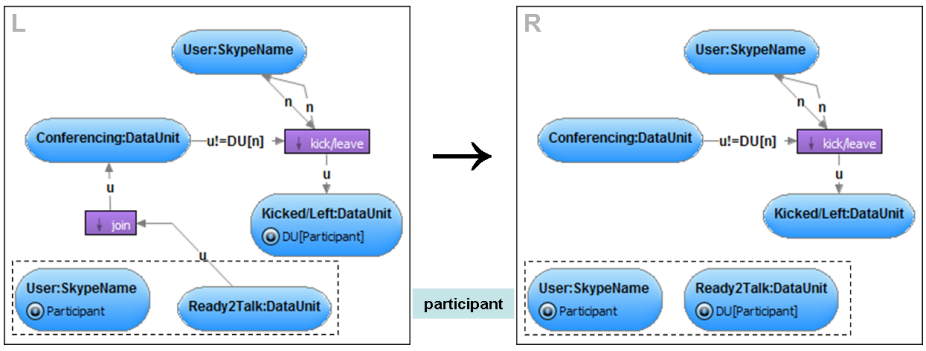


Fig. 7. Rule *KickParticipant* disconnecting participants who left a conference

4 Conclusion and Outlook

After introducing the general notion of Communication Spaces (CS), we discussed the new approach of AHOI nets as a formal modeling technique for CS. The examples, concerning particular features offered by Skype, show that this approach and the chosen modeling principles are powerful enough for first adequate modeling approach of CS. We have shown how we need to employ all the specialities of the AHOI nets as an integrated modeling technique:

- High-level data tokens are needed to represent data and user identities for effectively restricting communication.
- Moreover, higher-order tokens containing high-level nets are again used to represent users and their behavior and data in different system parts as conferences and chats [8].
- With reconfigurable Petri nets we enable the system to adapt itself to user requests and allow for a dynamic set of actors at system runtime. For more detail on the necessity of reconfiguration for modeling multicasting in dynamical Petri net systems we refer to [12].
- Petri nets with individual tokens [11] feature distinguishable tokens with identities, which are a technical premise to be able to formulate marking-changing rules. We have proven the existence of a weak adhesive high-level replacement system [6] for these nets, yielding a rich transformation theory.

To improve modeling usability, we plan to examine i.a. the following extensions of AHOI nets:

- An explicit control structure for defining rule sequences to ensure and enforce cleaning-up reconfigurations to be performed directly after the correspondent activity has terminated. Additionally, firing of transitions could be related to the application of particular rules and be comprised in the control structure.
- Amalgamated rule applications are similar to the "apply as long as possible" control structure, but in contrast, an amalgamated rule is applied to all

possible matches in the same step. With this technique, we can realize the schema of rule Transmit without explicitly formulating rules for all possible conference sizes. Moreover, we need this extension for non-local consistency-related concepts like Skype's Shared Groups, which are synchronized contact lists distributed between several clients. We contributed a first idea of how amalgamated rules can be used for multicasting data instead of the informal rule schema we used to formulate the rule *Transmit* in Fig. 5 [12].

Now that our formalism enables us to represent and simulate a model like Skype with the characterizing features of a CS, the most important part of our future work is to analyze and verify important properties of Skype in particular and to find/derive important properties we can prove for CS in general when using AHOI nets and the presented modeling approach with its principles in general. Examples of such properties are e.g. that a contact's owner must have confirmed all correspondent entries in his contact list, or that the system respects the clients' privacy settings when establishing communication via reconfiguration. To verify such properties we intend to make use of the rich theory of (high-level) Petri nets [4] and high-level replacement systems [6], especially the result for consistency and independency. We are currently elaborating the transformation theory of nets with individual tokens [11] by lifting the results of the collective approach and developing new analysis concepts.

To provide tool support for modeling and analyzing, we are also extending a graphical editor as an Eclipse plugin for a simplified kind of the AHL nets in [8] to support the AHOI formalism as we introduced it in this article.

References

1. Agha, G.: ACTORS: A Model of Concurrent Computation in Distributed Systems. PhD thesis, MIT (1985) Cambridge: MIT Press.
2. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press (June 1999)
3. Reisig, W.: Petri Nets: An Introduction. Volume 4 of EATCS Monographs on Theoretical Computer Science. (1985)
4. Jensen, K., Rozenberg, G., eds.: High-Level Petri Nets. (1991)
5. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Volume 6 of EATCS Monographs on Theoretical Computer Science., Berlin (1985)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. (2006)
7. Ehrig, H., Padberg, J., Ribeiro, L.: Algebraic high-level nets: Petri nets revisited. In: Proc. of the ADT-COMPASS Workshop'92 (Caldes de Malavella, Spain), Berlin (1993) Technical Report TUB93-06.
8. Hoffmann, K., Mossakowski, T., Ehrig, H.: High-Level Nets with Nets and Rules as Tokens. In: Proc. of 26th Intern. Conf. on Application and Theory of Petri Nets and other Models of Concurrency. Volume 3536 of LNCS. Springer Verlag (2005) 268–288
9. Valk, R.: Concurrency in communicating object petri nets. In: Concurrent Object-Oriented Programming and Petri Nets. (2001) 164–195

10. Ehrig, H., Hoffmann, K., Padberg, J., Ermel, C., Prange, U., Biermann, E., Modica, T.: Petri Net Transformations. In: Petri Net Theory and Applications. I-Tech Education and Publication (2008) 1–16
11. Ehrig, H., Modica, T., Biermann, E., Ermel, C., Hoffmann, K., Prange, U.: Low- and high-level petri nets with individual tokens. Technical Report 13/2009, Fakultät IV - Technische Unversität Berlin (2009) to appear.
12. Biermann, E., Ehrig, H., Ermel, C., Hoffmann, K., Modica, T.: Modeling Multicasting in Dynamic Communication-based Systems by Reconfigurable High-level Petri Nets. In: 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL-HCC'09). (2009)

Author Index