

## On the structural simplicity of machines and languages

Yael Moscovitz and Ehud Shapiro

Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel

E-mail: {yaels,udi}@wisdom.weizmann.ac.il

We employ an algebraic method for comparing the structural simplicity of families of abstract machines. We associate with each family of machines a first-order structure that includes machines as objects, composition operations, which construct larger machines from smaller ones, as functions, and a set of semantic relations. We then compare families of machines by studying the existence of homomorphisms between the associated structures. Given families of machines  $L$ ,  $L'$  with associated structures  $S$ ,  $S'$ , we say that  $L$  is *simpler* than  $L'$  if there is a homomorphism of  $S$  into  $S'$ , but not vice versa. We show that across several abstract machine models — finite automata, Turing machines, and logic programs — deterministic machines are simpler than nondeterministic machines and nondeterministic machines are simpler than alternating machines. Our results cross computational complexity boundaries. We show that for Turing machines, finite automata and logic programs every non-deterministic variant is not simpler than any deterministic variant, and that every alternating variant is not simpler than any nondeterministic (non-alternating) variant. In addition, we show that nondeterministic and alternating Turing machines are equivalent in their structural simplicity to iterative and ordinary logic programs, respectively, and that deterministic Turing machines are as simple as deterministic logic programs. We thus establish, in a sense somewhat independent of computational complexity classes and machine models, that determinism is simpler than nondeterminism and that nondeterminism is simpler than alternation. The method of comparison we employ is quite general. It has been applied successfully to the comparison of concurrent programming languages and we expect it to be applicable to other languages and machine models as well.

### 1. Introduction

Various methods have been proposed for the comparison of programming languages and models. Due to Turing-completeness of all programming languages, the effective methods are those that can introduce distinctions as well as similarities, relating the expressive power of the compared languages. Several methods succeed to compare only languages having a common basis (such as similar syntactic structure, similar semantic definition or similar evaluation method). Examples for such approaches can be found in [4, 7, 23], where sequential program schemes are compared, and in [2, 3, 6, 8, 9, 10, 11, 15, 17, 18, 19, 20, 24, 25, 29], where different

variants of languages and computational models are compared. Recently, several proposals have been made, in which a wider set of languages can be compared [21, 28, 30].

Shapiro [28] introduces a general method for comparing the structural simplicity of families of machines and languages. This method was successfully applied to compare a variety of different (not *a priori* related) concurrent programming languages [27, 28]. The generality of this method lies in associating a first-order structure with each language or family of machines, and in comparing languages (or families of machines) by demonstrating the existence or nonexistence of homomorphic mappings among their associated structures. The structure associated with a language or a family of machines has programs or machines as objects, a set of composition operations as functions, and a set of relations on programs. The composition operations are used to build larger machines from smaller ones, and the set of relations generally includes a semantic equivalence relation and/or its complement. The homomorphic mappings we investigate are called language embeddings. A language embedding is a homomorphism between the associated structures. A language  $L$  is considered *as simple as*  $L'$  if  $L$  can be embedded in  $L'$ . Furthermore,  $L$  is *simpler* than  $L'$  if there is an embedding of  $L$  in  $L'$ , but not *vice versa*. We use the following separation scheme (similar to the one introduced in [28]), in order to establish that an embedding between two languages does not exist.

**Separation scheme.** Let  $L$  and  $L'$  be languages with associated structures  $S$  and  $S'$ , respectively. If there exists a structure property  $\mathcal{P}$  such that:

1.  $\mathcal{P}$  is satisfied by  $S$ , but not by any homomorphic image of  $S$  in  $S'$ ,
2.  $\mathcal{P}$  is preserved by structure homomorphisms,

then  $L$  cannot be embedded in  $L'$ .

In this paper we further enrich the comparison method developed in [27, 28] and apply it to compare the simplicity of the deterministic, non-deterministic and alternating variants of finite automata, Turing machines and logic programming languages. (In logic programming languages the alternating variant is the language of ordinary logic programs, while the non-deterministic variant is the language of *iterative* logic programs — programs with at most one body-goal in every clause.) Although these models are not *a priori* related, we obtain both positive and negative results about the existence of language embeddings among them.

The structures associated with the models compared in this paper contain union as the sole composition operation. Thus, logic programs are composed to larger programs by taking the union of their clause sets, and machines are composed to larger machines by taking the union of their transition relations. Union as a sole composition operation is complete in the sense that all programs/machines can be built by taking the union of basic (smaller) programs/machines. Furthermore, it

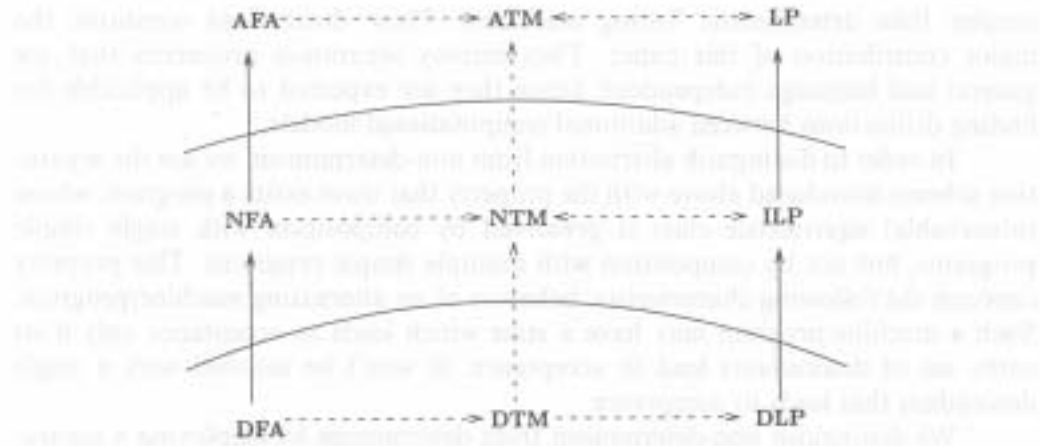


Fig. 1. Structure homomorphisms.

turns out that by employing this choice of composition operation, our comparison method successfully captures the intuitive differences between determinism, non-determinism and alternation, as demonstrated in the rest of this paper.

The semantic relations for the associated structures considered in this paper are taken from the set  $\{\simeq, \equiv, simple\}$ .  $\simeq$  is the observable equivalence relation,  $\equiv$  is the fully-abstract congruence relation (induced by  $\simeq$  and by the composition operations [13]), and *simple* is a unary predicate which identifies simple programs. Apart from the mathematical justification for this choice of semantic relations, we believe that the requirement that embeddings preserve the semantic equivalence relations is important since it ensures that the simplicity relations derived from our results preserve the meaning of the objects under comparison. In this sense, preserving the fully-abstract congruence relation is particularly important since its equivalence classes correspond to behaviors that can be detected by the external environment. The requirement to preserve the *simple* relation is more structural and we find it rather natural. It ensures that programs (or machines) with a very basic structure are mapped to the same kind of programs (or machines).

A summary of our results is presented in Figure 1. A curved line indicates that a homomorphism from every structure above it into any structure below it does not exist. An arrow from a structure *A* to *B* indicates that there exists a homomorphism of *A* into *B*. Solid arrows indicate homomorphisms which preserve the whole set of semantic relations:  $\{\simeq, \equiv, simple\}$ . Dashed arrows indicate homomorphisms, which preserve the observable equivalence relation ( $\simeq$ ); some of them preserve the *simple* relation as well.

The curved lines distinguish alternation from non-determinism and non-determinism from determinism, crossing computational complexity boundaries (for example, we show that alternating finite automata are not simpler than non-deterministic Turing machines, and that non-deterministic finite automata are not

simpler than deterministic Turing machines). These distinctions constitute the major contribution of this paper. They employ separation properties that are general and language independent; hence they are expected to be applicable for finding distinctions between additional computational models.

In order to distinguish alternation from non-determinism, we use the separation scheme introduced above with the property that there exists a program, whose (observable) equivalence class is preserved by composition with single simple programs, but not by composition with multiple simple programs. This property captures the following characteristic behavior of an alternating machine/program. Such a machine/program may have a state which leads to acceptance only if its *entire* set of descendants lead to acceptance. It won't be satisfied with a single descendant that leads to acceptance.

We distinguish non-determinism from determinism by employing a separation property describing a composition which is defined in the non-deterministic variants, but not in the deterministic variants. Such a composition captures the following typical situation which distinguishes non-deterministic models from deterministic ones. Non-deterministic models may introduce two possible computational paths that yield the same observable outcome, while in deterministic models the initial conditions uniquely determine a single possible computational path.

The arrows in Figure 1 describe our positive results. The vertical arrows represent inclusion mappings. As seen in the diagram, the inclusion mappings of finite automata are stronger (preserve more relations) than those of Turing machines, although these models are closely related. The proofs that the inclusion mappings of finite automata and logic programs (represented by solid arrows) preserve the fully-abstract congruence relation are not trivial and employ the notion of *testing sub-structures* [28]. The idea of testing sub-structures is a generalization of *experimenters* in process algebra presented in [16]. The main task in these proofs is to show that in each of the above inclusion mappings, the more restricted (source) structure is rich enough to detect programs (or automata) that are distinct under the fully-abstract congruence relation.

The horizontal arrows represent the known translations between logic programs and Turing machines [22, 26] and trivial embeddings of finite automata into Turing machines. Note that the relations between logic programs and Turing machines are obtained although they have different syntactic structure and employ different computation methods. The embeddings of finite automata in Turing machines and the embeddings of Turing machines in logic programs (all presented by dashed arrows) preserve the *simple* relation as well (in addition to the observational equivalence relation).

Some of the above results give mathematical justification to our intuitive notion about the differences and the similarities between the models investigated in this paper. For example, our negative results justify our intuitive notion about the difference in the expressive power of alternation, non-determinism and determinism. Furthermore, the separation properties that we use to obtain the above

negative results are quite general and give hope to obtain similar separations between deterministic, non-deterministic and alternating variants of other models, although such a generalization is neither trivial nor immediate. Among the positive results, the inclusion mappings between the different variants of the same model and the embeddings of finite automata in Turing machines are not surprising. The embeddings between logic programs and Turing machines and the difficulty to obtain the higher degree of comparison in several embeddings (*i.e.*, the difficulty to define embeddings that preserve the whole set of semantic relations) are more surprising and may give rise to a better understanding about the differences between the compared models.

The rest of the paper is organized as follows. In Section 2 we introduce the basic concepts of the comparison method presented in [27, 28] (slightly revised). In Section 3 we give a full description of the languages investigated in this paper. In Section 4 we introduce our (general) separation properties and apply them to distinguish alternation from non-determinism, and non-determinism from determinism, for all the models investigated in this paper. In Section 5 we present our positive results and claim that the involved mappings are homomorphic and preserve the semantic relations. Section 6 concludes the paper.

## 2. Framework

With every programming language or abstract machine model that we study, we associate a first-order structure, as follows.

**Definition 2.1 (Partial algebra).** A partial algebra is a pair  $\langle A; F \rangle$ , where  $A$  is a non-empty set and  $F$  is a set of finitary (possibly partial) operations on  $A$ .

**Definition 2.2 (First-order structure).** A first-order structure is a triple  $\langle P; F; R \rangle$  where  $\langle P; F \rangle$  is a partial algebra and  $R$  is a set of relations on  $P$ .

The first-order structure we associate with every language has the machines or programs with composition operations as algebras, and semantic relations which generally include a semantic equivalence relation. In our comparison method we investigate the existence of language embeddings between the compared languages. The notion of language embedding is based on finding homomorphic mappings between the associated structures.

**Definition 2.3 (Algebra homomorphism).** Let  $PA_1 = \langle A_1; F_1 \rangle$  and  $PA_2 = \langle A_2; F_2 \rangle$  be two partial algebras. A homomorphism  $\varepsilon$  of  $PA_1$  into  $PA_2$  is a mapping satisfying that for every  $a_1, \dots, a_n \in A_1$  and  $f \in F_1$  such that  $f(a_1, \dots, a_n)$  is defined,  $f \varepsilon(a_1 \varepsilon, \dots, a_n \varepsilon)$  is defined and satisfies  $(f(a_1, \dots, a_n)) \varepsilon = f \varepsilon(a_1 \varepsilon, \dots, a_n \varepsilon)$ .

**Definition 2.4 (Structure homomorphism).** Let  $S = \langle P; F; R \rangle$  and  $S' = \langle P'; F'; R' \rangle$  be

two first-order structures. A homomorphism  $\varepsilon$  of  $S$  into  $S'$  is a mapping satisfying:

1.  $\varepsilon$  is a homomorphism of  $\langle P; F \rangle$  into  $\langle P'; F' \rangle$  (as partial algebras).
2. For every  $a_1, \dots, a_n \in P$  and every  $n$ -ary relation  $r \in R$ , if  $r(a_1, \dots, a_n)$  holds in  $S$  then  $r\varepsilon(a_1\varepsilon, \dots, a_n\varepsilon)$  holds in  $S'$  (for every  $n$ ).

**Definition 2.5 (Faithful homomorphism).** Let  $S = \langle P; F; R \rangle$  and  $S' = \langle P'; F'; R' \rangle$  be two first-order structures. A homomorphism of  $S$  into  $S'$  is faithful if  $\forall r \in R, \neg r(\vec{a}) \implies \neg(r\varepsilon)(\vec{a}\varepsilon)$ .

Note that from a faithful homomorphism of  $\langle P; F; R \rangle$  into a structure  $\langle P'; F'; R' \rangle$ , one can derive a homomorphism of  $\langle P; F; R \cup \bar{R} \rangle$  into  $\langle P'; F'; R' \cup \bar{R}' \rangle$ .

We would like language embeddings to be defined for languages with different sets of operations. Furthermore, we would like embeddings to allow realizing an operation in the source language using an operation in the closure of operations in the target language. We therefore define the following:

**Definition 2.6 (Closure).** Let  $\langle A; F \rangle$  be an algebra. The closure of  $F$ , denoted by  $\widehat{F}$ , is the set of all finitary operations that correspond to "terms with variables" over  $F$  and  $A$  (the elements of  $A$  serve as nullary operations over  $A$ ).

For a structure  $S = \langle P; F; R \rangle$ , the closure of  $S$  is the structure  $\widehat{S} = \langle P; \widehat{F}; R \rangle$ .

**Definition 2.7 (Language embedding).** Let  $L$  and  $L'$  be languages with associated structures  $S$  and  $S'$ , respectively. A language embedding of  $L$  into  $L'$  is a homomorphism of  $S$  into  $\widehat{S}'$ .

The process of associating a first-order structure with a language is rather informal. It can be done in various ways, yielding various degrees of comparison between languages.

Let  $\mathcal{S}$  be a set of structures and let  $\mathcal{E}$  be a set of language embeddings among them, then  $\langle \mathcal{S}, \mathcal{E} \rangle$  constitutes a category which we call a *category of language embeddings*. A family  $\mathcal{K}$  of structure homomorphisms induces a preorder  $\preceq_{\mathcal{K}}$  between structures. For a family  $\mathcal{K}$  of structure homomorphisms, we denote by  $\mathcal{K}\mathcal{F}$  the family of all faithful homomorphisms in  $\mathcal{K}$ . Let  $S$  and  $S'$  be associated structures (of the languages  $L$  and  $L'$ , respectively).  $S \preceq_{\mathcal{K}} S'$  if and only if there is a  $\mathcal{K}$ -homomorphism of  $S$  in  $\widehat{S}'$ . We use  $S \sim_{\mathcal{K}} S'$  as a notation for  $S \preceq_{\mathcal{K}} S'$  &  $S' \preceq_{\mathcal{K}} S$ , and  $S \prec_{\mathcal{K}} S'$  as a notation for  $S \preceq_{\mathcal{K}} S'$  but  $S' \not\preceq_{\mathcal{K}} S$ . To prove that  $S \preceq_{\mathcal{K}} S'$ , one should introduce a  $\mathcal{K}$ -homomorphism of  $S$  in  $\widehat{S}'$ . To prove that  $S \not\preceq_{\mathcal{K}} S'$ , one should find a structure property  $\mathcal{P}$  and apply the following observation:

**Observation 2.1 (Separation scheme).** Let  $S$  and  $S'$  be associated structures of the languages  $L$  and  $L'$ , respectively. If there exists a structure property  $\mathcal{P}$  such that:

1.  $\mathcal{P}$  is satisfied by  $S$ , but not by any  $\mathcal{K}$ -homomorphic image of  $S$  in  $\widehat{S}'$ ,

2.  $\mathcal{P}$  is preserved by  $\mathcal{K}$ -homomorphisms,  
then  $S \not\leq_{\mathcal{K}} S'$ .

### 3. The computational models and their associated structures

We describe the models compared in this paper, and their associated structures. The objects in all structures are represented by tuples, in which every component is a set (as described below), and the single composition operation is defined to be component-wise union in all models.

The semantic relations employed in the associated structures of this paper are taken from the following relations:

- The observable equivalence relation, denoted  $\simeq$ , which characterizes when two machines/programs have the same externally observable behavior.
- The fully-abstract congruence relation, denoted  $\equiv$ , which is the (unique) coarsest congruence relation that respects the composition operation and is included in the observable equivalence relation. More specifically, for a first-order structure  $\langle P; F; R \rangle$  with  $\simeq \in R$ , the fully-abstract congruence relation,  $\equiv$ , is the unique coarsest relation satisfying:

$$p \equiv q \longrightarrow p \simeq q \text{ for every } p, q \in P$$

and

$$p_1 \equiv q_1, \dots, p_n \equiv q_n \longrightarrow f(p_1, \dots, p_n) \simeq f(q_1, \dots, q_n)$$

for every  $n$ -ary function  $f \in F$  and  $p_1, \dots, p_n, q_1, \dots, q_n \in P$ .

- A relation that identifies *simple* programs/machines. The predicate is defined specifically for each model. All simple programs/machines are observably equivalent to the empty program/machine.

The motivation behind the choice of union as the sole composition operation and the choice of the semantic relations were discussed in the introduction to this paper. We would like to further reflect the following property of the union operation and the semantic relations, which is used in the mathematical treatment of the structures described in this paper. Union is monotone in the sense that if a large program (or machine) is composed of two smaller programs/machines, every observable behavior obtained by the smaller programs (or machines) can be obtained by the large program or machine (composed of the smaller ones) as well.

Since the fully-abstract congruence is derived from the observable equivalence and the composition operations (union in our case), we specify below only the observable equivalence and the *simple* relation for every language.

Some of the languages compared in this paper are obtained by restricting the

programs under investigation. We use the notion of *sub-structure*, defined below, to describe the structures, associated with the restricted languages.

**Definition 3.1 (Sub-algebra).** Let  $\mathcal{A} = \langle A; F \rangle$  be a partial algebra, and let  $\emptyset \neq B \subseteq A$ .  $\mathcal{B} = \langle B; F \rangle$  is a relative sub-algebra of  $\mathcal{A}$  if for every  $f \in F$  and  $\vec{a} \in B^k$   $f(\vec{a})$  is defined and equals  $b$  in  $\mathcal{B}$  if and only if  $f(\vec{a}) = b$  is defined in  $\mathcal{A}$  and  $b \in B$ .

**Definition 3.2 (Sub-structure).** Let  $\langle A; F; R \rangle$  be a first-order structure, and let  $\emptyset \neq B \subseteq A$ .  $\langle B; F; R' \rangle$  is a sub-structure of  $\langle A; F; R \rangle$  if  $\langle B; F \rangle$  is a relative sub-algebra of  $\langle A; F \rangle$ , and every  $r' \in R'$  is a relation  $r \in R$ , restricted to  $B$ .

We first describe Turing machines and finite automata. We assume a predefined (infinite) set of states  $Q$  including a fixed initial state  $q_0 \in Q$ , a predefined (infinite) set of machine input symbols  $\Delta$  and a predefined set of tape symbols,  $\Gamma$  (for Turing machines only). It is therefore enough to specify in every machine only the transition relation  $\delta$ , the set of final states  $F \subseteq Q$ , and the set of universal states  $U \subseteq Q$  (for alternating machines only; in these machines  $F \subseteq U$ ).

Since we want to include in our compared models the families of alternating Turing machines and alternating finite automata, with union as a composition operation, we would like the union operation to be monotonic in the sense that bigger machines accept bigger languages. We therefore replace the definition of the transition relation  $\delta$  by an operationally equivalent definition. We define the successor of every state to be a *set* of states instead of a single state, where existential states (states in  $Q - U$ ) may have only singletons as their successors. A universal state may introduce also an existential behavior in the sense that it has more than one set of successors. A pure universal state is a state in  $U$  that has at most one set of successors. As a result, the union of Turing machines, having universal states with identical state names, increases the non-determinism in the composed machine. (Unlike the previous definition, where such a union reduced the accepted language.)

We now describe the structures, associated with the models compared in this paper. In all structures, we use  $R$  as the set of relations  $\{\simeq, \equiv, \text{simple}\}$ .

#### Turing machines:

$ATM = \langle ATM; \cup; R \rangle$ :  $ATM$  is the set of alternating Turing machines, as presented by Chandra *et al.* [5] and by Fischer and Ladner [12].

A typical  $ATM$  element is  $M = \langle \delta, U, F \rangle$ , where:

$\delta$  is a set of quintets of the form  $(q, a, Q', d', LR)$ , meaning that upon reading the character  $a$  in the state  $q$  the head writes  $d'$  instead, moves either to the left or to the right according to the value of  $LR$ , and execution continues, with the states in  $Q'$  as next states.

If  $M_1$  and  $M_2$  are Turing machines,  $M_1 \simeq M_2$  if and only if  $L(M_1) = L(M_2)$ , where  $L(M_i)$  is the language accepted by  $M_i$  ( $i = 1, 2$ ).

A machine is *simple* if it has the form  $(\emptyset, \{q\}, \{q\})$  for some  $q \in Q$ .

$\mathcal{NTM} = \langle \mathcal{NTM}; \cup; R \rangle$  is the sub-structure of  $\mathcal{ATM}$ , determined by  $\mathcal{NTM}$ , the subset of nondeterministic Turing machines.

$\mathcal{DTM} = \langle \mathcal{DTM}; \cup; R \rangle$  is the sub-structure of  $\mathcal{NTM}$  determined by  $\mathcal{DTM}$ , the subset of deterministic Turing machines.

#### Finite automata:

$\mathcal{AFA} = \langle \mathcal{AFA}; \cup; R \rangle$ :  $\mathcal{AFA}$  is the set of alternating finite automata with a typical element  $(\delta, U, F)$ , where:

$\delta$  is a set of triples of the form  $(q, a, Q')$ , meaning that upon reading the character  $a$  in the state  $q$  execution continues, with the states in  $Q'$  as next states. Observable equivalence and the *simple* relation, as well as the non-deterministic and the deterministic structures ( $\mathcal{NFA}$  and  $\mathcal{DFA}$ ) are defined analogously to Turing machines.

#### Logic program modules:

A *logic program* is a set of definite clauses. A *definite clause* is a clause of the form  $p \leftarrow q_1, \dots, q_n$  where  $p, q_1, \dots, q_n$  are atoms.  $p$  is the clause head and  $q_1, \dots, q_n$  is the clause body. An *iterative* clause is a clause in which  $n = 1$ , and a *unit* clause is a clause with an empty body ( $n = 0$ , logically corresponding to the atom *true* as its body). A *deterministic* logic program is a program in which all clause heads are pairwise non-unifiable and every clause variable appears in its head. A *logic program module* is a logic program, associated with a specification of the set of its exported (externally visible) predicate names. The structure associated with the language of logic program modules is defined as follows.

$\mathcal{LP} = \langle \mathcal{LP}; \cup; R \rangle$ :  $\mathcal{LP}$  is the set of logic program modules of the form  $(P, E)$ , where  $P$  is a logic program and  $E$  is a set of exported predicate names (import/export clauses with an empty set of imported clauses in the notion of [13, 14]).

If  $(P_1, E_1)$  and  $(P_2, E_2)$  are logic program modules,  $(P_1, E_1) \simeq (P_2, E_2)$  if and only if  $[[P_1]]|_{E_1} = [[P_2]]|_{E_2}$ , where  $[[P_i]]|_{E_i}$  is the set of ground atoms derived from  $P_i$ , restricted to atoms whose predicates are in  $E_i$  ( $i = 1, 2$ ).

A logic program module is *simple* if it has the form  $(\{C\}, \emptyset)$ , where  $C$  is a unit clause.

$\mathcal{ILP}$  and  $\mathcal{DLP}$  are the sub-structures of  $\mathcal{LP}$ , determined by  $\mathcal{ILP}$  (the set of iterative logic program modules) and by  $\mathcal{DLP}$  (the set of deterministic logic program modules), respectively.

#### 4. Negative results: Separation

In this section we distinguish determinism from non-determinism, and non-

determinism from alternation, crossing computational complexity boundaries. We show that in the models investigated in this paper, every alternating model is not simpler than any non-deterministic one, which is not simpler than any deterministic one.

We present two (abstract) separation properties which, together with the separation scheme, supply the distinctions between the models. These properties refer to associated structures having the set  $\{\simeq, \equiv, \text{simple}\}$  as semantic relations. With the help of the separation properties presented in this section, we can classify pairs of such structures, between which faithful homomorphisms do not exist. The first property introduces a program, whose (observable) equivalence class is preserved by composition with single simple programs, but not by composition with multiple simple programs. We use it to conclude that every alternating language is not simpler than any non-deterministic (non-alternating) language. The second property introduces a program  $q$  having two different, observably equivalent, compositions with the simple programs  $q_1$  and  $q_2$ , satisfying that the composition of  $q$  with both  $q_1$  and  $q_2$  is a well-defined program. We use it in order to conclude that every non-deterministic language is not simpler than any deterministic language.

It is important to realize that although the separation properties introduced in this section are close to our intuitive notion about the differences between determinism, non-determinism and alternation, the task of formulating these properties is certainly not trivial. The separation properties should be formulated in terms of the structures components and obey the mathematical requirements introduced by the separation scheme. Furthermore, an additional effort was made in order to obtain the separation properties introduced in this section and make them rather general. Thus, each of them is powerful enough to provide us with nine separation results (consisting of all the pairs of deterministic and non-deterministic models and all the pairs of non-deterministic and alternating models investigated in this paper). Due to this nature of our separation properties, we believe that they really capture the differences between the models investigated in this paper. However, a further evidence for the non-trivial task of finding separation properties is that although we believe that the same ideas may help us to distinguish between other deterministic, non-deterministic and alternating models, applying these properties to other models (like [1]) is not immediate. It requires a closer examination of their formulation and their relation to the structures under consideration.

#### 4.1. PRELIMINARIES

The separation properties introduced in this section refer to associated structures which have  $\oplus$  as an operation and the set  $\{\simeq, \equiv, \text{simple}\}$  as relations, with *simple* being a unary relation and  $\equiv$  being the fully-abstract equivalence relation induced by  $\simeq$  and  $\oplus$ . We denote by  $\mathcal{S}$  the set of all such structures. We denote by  $\mathcal{F}$  the family of homomorphisms between  $\mathcal{S}$ -structures.

**Notation:** For a structure  $L$  we denote by  $\text{simple}(L)$  the set of all simple objects in  $L$  (the programs or machines for which the unary relation *simple* holds). We denote by  $s^L$  the equivalence class (under  $\simeq$ ) of all the simple objects in  $L$ .

**Definition 4.1 ( $n$ -modification).** Let  $\langle P; F; R \rangle$  be a first order structure in  $\mathcal{S}$  and let  $q \in P$  be a program/machine. We say that  $q$  is  $n$ -modified by the set of programs/machines  $P_1$  if there exist  $q_1, \dots, q_n \in P_1$  such that  $q \oplus q_1 \oplus \dots \oplus q_n \neq q$ .

**Definition 4.2 (Bi-modification).** Let  $L = \langle P; F; R \rangle$  be a first-order structure in  $\mathcal{S}$  and let  $q \in P$  be a program/machine. We say that  $q$  is bi-modified by the  $L$ -programs/machines  $q_1$  and  $q_2$  if  $q_1 \neq q_2$ ,  $q \oplus q_i \neq q$  (for  $i = 1, 2$ ) and  $q \oplus q_1 \simeq q \oplus q_2$ .

#### 4.2. MODIFICATION: SEPARATING BETWEEN ALTERNATION AND NON-DETERMINISM

We use the following structure property in order to distinguish alternation from non-determinism.

**Property 4.1 (No 1-modification).** For the structure  $L$ , there exists an object  $q$  in  $L$  such that  $q \simeq s^L$ ,  $q$  is  $n$ -modified by  $\text{simple}(L)$  for some  $n > 1$ , but  $q$  is not 1-modified by  $\text{simple}(L)$ .

From the following propositions and Observation 2.1 (the separation scheme), we conclude that for the structures investigated in this paper, every structure associated with a non-deterministic (non-alternating) language cannot embed any structure associated with an alternating language, using  $\mathcal{FF}$ -homomorphisms.

**Proposition 4.1.** Property 4.1 is preserved by  $\mathcal{FF}$ -homomorphisms.

*Proof.* Let  $\varepsilon : L \rightarrow L'$  be an  $\mathcal{FF}$ -homomorphism. Let  $q$  be an object in  $L$  satisfying  $q \simeq s^L$ ,  $q$  is  $n$ -modified by  $\text{simple}(L)$  ( $n > 1$ ) but  $q$  is not 1-modified by  $\text{simple}(L)$ . We show that  $q\varepsilon \in L'$  satisfies the same conditions. By the assumptions on  $q$  and since  $\varepsilon$  is faithful,  $q\varepsilon \simeq s^{L'}$ . In addition, it is easy to see that since  $\varepsilon$  is an  $\mathcal{FF}$ -homomorphism (hence it preserves the observable equivalence and its complement and maps (non-)simple objects to (non-)simple objects),  $q\varepsilon$  is  $n$ -modified ( $n > 1$ ), but not 1-modified, by  $\text{simple}(L')$ . ■

**Proposition 4.2.** Let  $L \in \{\mathcal{LP}, \mathcal{ATM}, \mathcal{AFA}\}$ . There exists an object  $p$  in  $L$  such that  $p \simeq s^L$  (thus  $\text{obs}(p) = \emptyset$ ),  $p$  is  $n$ -modified by  $\text{simple}(L)$  for some  $n > 1$ , but not 1-modified by  $\text{simple}(L)$ .

*Proof.* For  $\mathcal{LP}$  we take the program  $(\{p \leftarrow q, r, s\}, \{p\})$ . This program has an empty set of observables. It is modified by  $(\{q\}, \emptyset) \cup (\{r\}, \emptyset) \cup (\{s\}, \emptyset)$ , but it cannot be modified by any single simple program.

For  $\mathcal{ATM}$  and  $\mathcal{AFA}$  we similarly take a machine having a transition from the initial state  $q_0$  to the universal state  $p$  and an (alternating) transition from  $p$  to the set  $\{q, r, s\}$  of different states (no other transitions are employed). This machine is observably equivalent to the empty machine. It can be modified by the machine  $M_q \cup M_r \cup M_s$  where for a state  $f$ ,  $M_f$  is the simple machine containing only the initial state  $q_0$  and the final state  $f$  (no transitions). However, it cannot be modified by any single simple machine. ■

**Proposition 4.3.** *Let  $L_1 \in \{\mathcal{LP}, \mathcal{ATM}, \mathcal{AFA}\}$  and let  $L_2 \in \{\mathcal{ILP}, \mathcal{NTM}, \mathcal{NFA}\}$ . Let  $L$  be an  $\mathcal{FF}$ -homomorphic image of  $L_1$  in  $L_2$ . Let  $p$  in  $L$  be a program/machine satisfying  $p \simeq s^\perp$  (thus  $\text{obs}(p) = \emptyset$ ). If  $p$  is  $n$ -modified by  $\text{simple}(L)$  (for some  $n > 0$ ) then  $p$  is 1-modified by  $\text{simple}(L)$ .*

*Proof.* Since  $p$  is  $n$ -modified by  $\text{simple}(L)$ , the graph derived from the machine  $p$  (in the case of  $\mathcal{NTM}$  and  $\mathcal{NFA}$ ) is not empty. Analogously in  $\mathcal{ILP}$   $p \neq (\emptyset, \emptyset)$ . (Otherwise,  $p$  composed with the  $n$  simple program that modify it will be observably equivalent to the composition of the  $n$  simple programs. But the composition of the  $n$  simple programs should be observationally equivalent to a single simple program while the first composition is not, due to  $n$ -modification).

Moreover, in the case of  $\mathcal{NTM}$  and  $\mathcal{NFA}$  we can further conclude that in the graph derived from  $p$  there is a non-trivial path (or several paths), which is extended by simple machines to form a path (or several paths) from  $q_0$  to a final state. Since  $p$  is non-alternating, all paths are real simple paths (threaded) and therefore it is enough to extend *one* path in order to obtain modification. Hence,  $p$  is 1-modified by  $\text{simple}(L)$ .

In the case of  $\mathcal{ILP}$  the program  $p$  can have either the form  $(\emptyset, G)$  or  $(P, G)$ . In the former case, a simple program of the form  $(\{p\}, \emptyset)$ , where  $p$  is one of the predicates in  $G$ , is enough to obtain 1-modification. In the latter case, similar to the case of machines, due to the modification of  $p$  by  $\text{simple}(L)$ ,  $p$  should contain a threaded program which can be composed with a *single* simple program in order to obtain modification. Therefore,  $p$  is 1-modified by  $\text{simple}(L)$ . ■

**Corollary 4.1.** *For  $L \in \{\mathcal{LP}, \mathcal{ATM}, \mathcal{AFA}\}$  and  $L' \in \{\mathcal{ILP}, \mathcal{NTM}, \mathcal{NFA}\}$ ,  $L \not\leq_{\mathcal{FF}} L'$*

#### 4.3. COMPOSITION OF BI-MODIFIED PROGRAMS: SEPARATING BETWEEN NON-DETERMINISM AND DETERMINISM

We use the following structure property in order to distinguish non-determinism from determinism.

**Property 4.2 (Composition and bi-modification).** *For a structure  $L = (Pr, F; R) \in \mathcal{S}$*

with  $\oplus \in F$ , there exists an object  $p \in Pr$  satisfying  $p \simeq s^L$ , there exist simple objects  $p_1, p_2 \in Pr$  that bi-modify  $p$  and the composition  $p \oplus p_1 \oplus p_2$  is defined in  $L$ .

From the following propositions and Observation 2.1 (the separation scheme), we conclude that for the structures investigated in this paper, every structure associated with a non-deterministic language cannot be embedded in any structure associated with a deterministic one, using  $\mathcal{FF}$ -homomorphisms.

**Proposition 4.4.** *Property 4.2 is preserved by  $\mathcal{FF}$ -homomorphisms.*

*Proof.* Let  $\varepsilon : L \rightarrow L'$  be an  $\mathcal{FF}$ -homomorphism. We show that if Property 4.2 is satisfied by  $L$ , then it is satisfied by  $L\varepsilon$  as well. Let  $p, p_1, p_2 \in L$  be the objects that satisfy Property 4.2. We show that  $p\varepsilon, p_1\varepsilon$  and  $p_2\varepsilon$  satisfy Property 4.2.  $p\varepsilon \in L\varepsilon$  satisfies  $p\varepsilon \simeq s^{L\varepsilon}$  since  $\varepsilon$  is an  $\mathcal{FF}$ -homomorphism (in particular it is faithful with respect to the observable equivalence and the *simple* relations).

$p_1\varepsilon$  and  $p_2\varepsilon$  are simple since  $p_1$  and  $p_2$  are and  $\varepsilon$  is an  $\mathcal{FF}$ -homomorphism.  $p_1\varepsilon$  and  $p_2\varepsilon$  bi-modify  $p\varepsilon$  since  $p_1$  and  $p_2$  bi-modify  $p$  and it is easy to check that bi-modification is preserved by  $\mathcal{FF}$ -homomorphisms (which in particular are faithful with respect to the observable equivalence and the fully-abstract congruence relations). Finally, the composition  $p\varepsilon \oplus p_1\varepsilon \oplus p_2\varepsilon$  is defined in  $L\varepsilon$  because since  $\varepsilon$  is homomorphic it is equal to  $(p \oplus p_1 \oplus p_2)\varepsilon$  and  $p \oplus p_1 \oplus p_2$  is defined in  $L$ . ■

**Proposition 4.5.** *Let  $L \in \{\mathcal{ICP}, \mathcal{NTM}, \mathcal{NFA}\}$ . There exists a program/machine  $p \in L$  and  $p_1, p_2 \in \text{simple}(L)$  such that  $\text{obs}(p) = \emptyset$  (thus  $p \simeq s^L$ ), the programs/machines  $p_1, p_2$  bi-modify  $p$  and the composition  $p \cup p_1 \cup p_2$  is defined in  $L$ .*

*Proof.* For  $\mathcal{ICP}$  we take  $p$  to be the program  $(\{q \leftarrow a_1, q \leftarrow a_2\}, \{q\})$ ,  $p_i$  is the program  $(\{a_i\}, \emptyset)$  ( $i = 1, 2$ ).

For  $\mathcal{NTM}$  and  $\mathcal{NFA}$  we similarly take  $p$  to be a machine that includes the states  $q_0, q_1, q_2$  and a transition from  $q_0$  to  $q_i$  ( $i = 1, 2$ ) upon reading some character  $a$  (the Turing machine writes  $a$  and moves to the right in these transitions). The set of final states in  $p$  is empty. We take  $p_i$  to be the machine including the initial state  $q_0$  and a single final state  $q_i$  (no transitions).  $p_1$  and  $p_2$  bi-modify  $p$  ( $p \cup p_i$  accept the language containing the single word  $a$  in the case of finite automata, and all the words beginning with  $a$  in the case of Turing machines). In addition  $p \cup p_1 \cup p_2$  is defined since composition is total in these models. ■

**Proposition 4.6.** *Let  $L_1$  be a structure in  $\{\mathcal{ICP}, \mathcal{NTM}, \mathcal{NFA}\}$ , let  $L_2$  be a structure in  $\{\mathcal{DCP}, \mathcal{DTM}, \mathcal{DFA}\}$ , and let  $L$  be an  $\mathcal{FF}$ -homomorphic image of  $L_1$  in  $\widehat{L_2}$ . Let  $p$  be an  $L$ -object satisfying  $p \simeq s^L$  and let  $p_1, p_2$  be simple  $L$ -objects that bi-modify  $p$ . Then  $p \cup p_1 \cup p_2$  is undefined in  $L$ .*

*Proof.* Suppose  $L_2 = \mathcal{NTM}$ ,  $p \in L$  is a Turing machine satisfying  $\text{obs}(p) = \emptyset$  and

$p_1, p_2 \in L$  are simple Turing machines that bi-modify  $p$ . We first observe that the bi-modification is obtained due to the simple machines  $p_1$  and  $p_2$  and not due to the composition operation, since otherwise, the image of an empty machine is found to modify  $p$  and this leads to a contradiction, since the observable equivalence should be preserved by  $\mathcal{FF}$ -homomorphisms. Therefore, there exists a path in  $p$  which is extended by  $p_1$  to form a path from  $q_0$  to a state  $q_1$ , where  $q_1$  is the final state in the simple machine  $p_1$ . ( $q_1 \neq q_2$  since  $p_1 \neq p_2$ .) We want to show that  $p^* = p \cup p_1 \cup p_2$  is non-deterministic. By definition, the successful computations of both  $p \cup p_1$  and  $p \cup p_2$  are successful computations of  $p \cup p_1 \cup p_2$ . Since  $p \cup p_1 \simeq p \cup p_2$ , the initial configurations of the successful computations of  $p \cup p_1$  are identical to those of  $p \cup p_2$  (since they have the same words written on the tapes). Thus,  $p^*$  includes two successful computations, having the same initial configuration, but one has a final configuration that contains the state  $q_1$  and the other has a final configuration containing the final state  $q_2$ . Assume by contradiction that  $p^*$  is deterministic. Then, it can be shown, by induction on the computation length, that starting from a fixed initial configuration there is a unique final configuration. (Inductively, given a fixed initial configuration, for every  $n \geq 0$  there is a unique prefix of a successful computation whose length is  $n$ .) This contradicts the existence of two final configurations, one contains  $q_1$  and the other contains  $q_2$ . Therefore  $p^*$  is non-deterministic.

In finite automata, using similar arguments, the successful computations of  $p \cup p_1$  are successful computations of  $p^*$ . For every word accepted by a deterministic automaton, we can show, by induction on the length of the accepted word, that the deterministic automaton has a unique computation accepting this word. Therefore, since  $p \cup p_1 \simeq p \cup p_2$ , if we assume that  $p^*$  is deterministic, we get a contradiction, similar to the case of Turing machines.

Similarly, in logic programs, we show by induction on the computation length, that an initial ground goal has a unique successful derivation using a deterministic program. Therefore, using similar arguments, if  $p$  (satisfying  $obs(p) = \emptyset$ ) is bi-modified by the two (different) simple programs  $p_1$  and  $p_2$ , then  $p \cup p_1 \cup p_2$  is non-deterministic. ■

**Corollary 4.2.** For  $L \in \{ILP, NTM, NFA\}$  and  $L' \in \{DLP, DTM, DFA\}$ ,  
 $L \not\leq_{\mathcal{FF}} L'$

## 5. Positive results: Embeddings

In this section we present positive results about existing embeddings between the structures. We introduce  $\mathcal{OF}$ -homomorphisms and  $\mathcal{FF}$ -homomorphisms. An  $\mathcal{OF}$ -homomorphism is a faithful homomorphism between structures whose set of semantic relations consists of the singleton set  $\{\simeq\}$ . An  $\mathcal{FF}$ -homomorphism is a faithful homomorphism between structures whose set of semantic relations consists of the set  $\{\simeq, \equiv, simple\}$ .

## 5.1. SUBSET EMBEDDINGS

By employing language embeddings, which are inclusion mappings, we obtain the following results. These results relate sub-structures to their extended structures.

**Theorem 5.1**

1.  $DLP \preceq_{FF} ILP \preceq_{FF} LP$
2.  $DFA \preceq_{FF} NFA \preceq_{FF} AFA$

*Proof.* As restrictions of the identity relation, the above embeddings trivially preserve the observable equivalence and the *simple* relations. Furthermore, we use Observation 5.1 below (taken from [28]) to conclude from Propositions 5.1–5.4 (below) that the above homomorphisms preserve the fully-abstract congruence relation as well. ■

In the following, we define the notion of a *testing sub-structure* and use it in order to prove that the embeddings of Theorem 5.1 are faithful with respect to the fully-abstract congruence relation. The notion of testing sub-structures is related to the notion of testing equivalence, presented for process algebra in [16] as follows. In [16], two processes (analogous to programs or machines in our context) are defined to be equivalent if they introduce the same behavior when composed with a predefined set of experimenters. According to our definitions, two programs (or machines) in  $L$  are equivalent under the fully-abstract congruence relation if they introduce the same observable behavior when composed with programs (or machines) in  $L$ . In the terminology of [16] the whole set of objects in  $L$  serve as experimenters for testing equivalence. Thus, in order to prove that an embedding is faithful with respect to the fully-abstract congruence, one should show that (non-) equivalent objects in the source structure are mapped to (non-) equivalent object in the closure of the target structure (here we refer to equivalence under the fully-abstract congruence relation). Testing sub-structures enables us to reduce the set of experimenters to objects of the testing sub-structure rather than to the whole target structure. It turns out that if the image embedding is a testing sub-structure for the target structure (*i.e.*, the set of experimenters can be reduced to the embedding image), an embedding which is faithful with respect to the observable equivalence relation is also faithful with respect to the fully-abstract congruence relation. In particular, in the case of subset embeddings, we show that the source (more restricted) structure is a testing sub-structure for the target (extended) structure. The full details are given below.

**Definition 5.1 (Context).** Let  $A = \langle P; F \rangle$  be a partial algebra.  $C(x)$  is a context of  $A$  if it is a term over  $P$  (as nullary operations) and  $F$  (as operations) with a single variable  $x$ .

**Definition 5.2 (Testing sub-structure).** Let  $S = \langle P; F; R \rangle$  be a structure where  $\{\simeq, \equiv\} \subseteq R$ , with  $\equiv$  being the fully-abstract equivalence induced by  $\simeq$  and  $F$ . A sub-structure  $T$  of  $\widehat{S}$  is a testing sub-structure of  $\widehat{S}$  if for every  $q, r \in P$  such that  $q \not\equiv r$  there exists a  $T$ -context,  $C(x)$ , satisfying  $C(q) \not\equiv C(r)$ .

**Observation 5.1.** Let  $\varepsilon$  be a faithful structure homomorphism of  $S = \langle P; F; R \rangle$  into  $S' = \langle P'; F'; R' \rangle$ , where  $\simeq \in R$  and  $\{\simeq, \equiv\} \subseteq R'$ . If  $S\varepsilon$  is a testing sub-structure of  $S'$  then  $\varepsilon$  constitutes a faithful homomorphism of  $\langle P; F; R \cup \{\equiv\} \rangle$  into  $S'$ .

**Proposition 5.1.** For every set of semantic relations  $R$ ,  $\langle ILP; \cup; R \cup \{\simeq, \equiv\} \rangle$  is a testing sub-structure for  $\langle LP; \widehat{\cup}; R \cup \{\simeq, \equiv\} \rangle$ .

*Proof.* Assume  $(P, G_P), (Q, G_Q) \in \mathcal{LP}$  such that  $(P, G_P) \simeq (Q, G_Q)$  but  $(P, G_P) \not\equiv (Q, G_Q)$  (if  $(P, G_P) \not\equiv (Q, G_Q)$  then the identity context distinguishes them). Let  $(R, G_R) \in \mathcal{LP}$  be a program that distinguishes  $(P, G_P)$  from  $(Q, G_Q)$ ; hence  $(P, G_P) \cup (R, G_R) \not\equiv (Q, G_Q) \cup (R, G_R)$  (we can assume that the distinguishing context has this form since it is a context built in a structure having a sole composition operation). Without loss of generality, assume that there is a goal  $G$  which is derived from  $(P, G_P) \cup (R, G_R)$  and not from  $(Q, G_Q) \cup (R, G_R)$ .

We now claim:

**Lemma 5.1.** Let  $G$  be a goal which is derived from  $(P, G_P) \cup (R, G_R)$  and not from  $(Q, G_Q) \cup (R, G_R)$  (using the derivation rules of the Appendix). There exists an iterative program  $R^I \in ILP$  such that  $G$  is derived from  $(P, G_P) \cup (R^I, G_R)$  and not from  $(Q, G_Q) \cup (R^I, G_R)$ .

Due to the above lemma, the iterative program  $(R^I, G_R)$  distinguishes  $(P, G_P)$  from  $(Q, G_Q)$ , and hence  $\langle ILP; \cup; R \cup \{\simeq, \equiv\} \rangle$  is a testing sub-structure for  $\langle LP; \widehat{\cup}; R \cup \{\simeq, \equiv\} \rangle$ .

We prove the lemma by induction on the number of reductions with non-iterative clauses of  $R$  in the derivation of  $G$  from  $P \cup R$ . If all the clauses in this derivation are iterative, then  $R^I$ , which is the subset of  $R$  including only the clauses employed in the derivation of  $G$  from  $P \cup R$ , satisfies the requirements of the lemma.

Assume the lemma holds for every goal whose derivation requires at most  $k$  reductions with non-iterative clauses of  $R$ . Let  $G$  be a goal whose derivation from  $P \cup R$  employs  $k+1$  reductions with non-iterative clauses of  $R$ . Let  $C \in R$  be the first non-iterative clause of  $R$  employed in the derivation of  $G$ . Thus,  $A$  is derived from (the already derived goals)  $B_1, \dots, B_n$ , where  $A \leftarrow B_1, \dots, B_n$  is an instance of  $C$ .

If  $B_1, \dots, B_n$  can all be obtained in a derivation of  $(Q \cup R, G_Q \cup G_R)$  then let  $R'$  be  $R \cup \{A\}$ . In  $P \cup R'$  there is a derivation for  $G$ , employing at most  $k$  non-iterative clauses of  $R'$  (since the new added clause can be used for  $A$  instead of

C). Furthermore,  $G$  cannot be derived from  $(Q, G_Q) \cup (R', G_R)$  — otherwise it would have been derived from  $(Q, G_Q) \cup (R, G_R)$ . Therefore, by the induction hypothesis, there exists a program  $R^l$  which satisfies the conditions of the lemma.

Otherwise there exists a goal  $B_i$ , obtained in the derivation of  $G$  from  $(P, G_P) \cup (R, G_R)$  as described above, which cannot be obtained in a derivation of  $(Q \cup R, G_Q \cup G_R)$ . Thus,  $G$  can be derived from  $(P \cup R \cup \{A \leftarrow B_i\}, G_P \cup G)$  by using at most  $k$  iterative clauses of  $R \cup \{A \leftarrow B_i\}$  (employing  $A \leftarrow B_i$  instead of  $C$ ). Furthermore,  $G$  cannot be derived from  $(Q \cup R \cup \{A \leftarrow B_i\}, G_Q \cup G_R)$  due to the preconditions of the lemma and due to the choice of  $B_i$ . Hence, by the induction hypothesis, there is an *iterative* program  $R^l$  which satisfies the required conditions. ■

**Proposition 5.2.** *For every set of semantic relations  $R, \langle DLP; \cup; R \cup \{\simeq, \equiv\} \rangle$  is a testing sub-structure for  $\langle ILP; \hat{\cup}; R \cup \{\simeq, \equiv\} \rangle$ .*

*Proof.* Assume  $(P, G_P), (Q, G_Q) \in ILP$  such that  $(P, G_P) \simeq (Q, G_Q)$  but  $(P, G_P) \not\equiv (Q, G_Q)$ . Let  $(R, G_R) \in ILP$  be a program that distinguishes  $(P, G_P)$  from  $(Q, G_Q)$ ; hence  $(P, G_P) \cup (R, G_R) \not\equiv (Q, G_Q) \cup (R, G_R)$ . Without loss of generality, assume that there is a ground goal  $G$  which is derived from  $(P, G_P) \cup (R, G_R)$  and not from  $(Q, G_Q) \cup (R, G_R)$ . We construct a deterministic program  $R^D$  such that  $(R^D, G_R)$  distinguishes between  $(P, G_P)$  and  $(Q, G_Q)$  by satisfying the same derivation relation as  $R$  with respect to  $G$ .

We first define the following:

**Definition 5.3 (Ground substitution).** *Let  $C$  be a non-ground clause (or atom) in the derivation of  $G$  from  $P$ . Let  $a$  be a constant, not appearing in the derivation of  $G$  from  $P$ . Assume  $X_1, \dots, X_m$  is the list of variables in  $C$ .  $\theta_G^{(C,a)}$  is defined to be the substitution  $\{X_1 \leftarrow a, \dots, X_m \leftarrow a\}$ .*

We construct  $R^D$  by traversing the derivation of  $G$  from  $(P, G_P) \cup (R, G_R)$ . For every (unquantified) clause  $C$  in the derivation, which is an instance of a clause in  $R$ , we include  $C\theta_G^{(C,a)}$  to be a clause in  $R^D$  as long as a clause with an identical head has not yet been included in  $R^D$ . The resulting program,  $R^D$  is deterministic since all its clauses are ground and clause heads are different from each other. The proposition now follows from the following lemmas:

**Lemma 5.2.** *Let  $A$  be an atom in the derivation of  $G$  from  $(P, G_P) \cup (R, G_R)$ . Then  $A\theta_G^{(A,a)}$  has a derivation from  $P \cup R^D$ .*

*Proof.* We prove the lemma by induction on the position of  $A$  in the derivation of  $G$  from  $(P, G_P) \cup (R, G_R)$ . The lemma holds trivially if  $A$  is the atom *true* which is always derivable.

Otherwise,  $A$  is derived from previous expressions, using one of the derivation

rules described in the Appendix. Hence, there is a clause  $A \leftarrow B$ , which is an instance of a clause in  $P \cup R$ , and this clause instance appears in the derivation prior to  $A$ . If the above clause is an instance of a clause in  $P$ , then by the derivation definition every atom in  $B$  appears in the derivation prior to  $A$  and hence, by the induction hypothesis every atom in  $B\theta_G^{(B,a)}$  has a derivation from  $P \cup R^D$ . Therefore  $A\theta_G^{(A,a)}$  has a derivation from  $P \cup R^D$  by using the clause instance  $(A \leftarrow B)\theta_G^{(A \leftarrow B,a)}$  and the MP-rule.

Otherwise assume  $A \leftarrow B$  is an instance of a clause in  $R$ . If a clause  $A' \leftarrow B'$  satisfying  $A'\theta_G^{(A',a)} = A\theta_G^{(A,a)}$ , appears in the derivation prior to  $A \leftarrow B$  then, by the derivation definition, the atom  $A'$  should appear in the derivation prior to  $A$ . By the induction hypothesis  $A'\theta_G^{(A',a)}$  can be derived from  $P \cup R^D$ . The same is true for  $A\theta_G^{(A,a)}$  since  $A\theta_G^{(A,a)} = A'\theta_G^{(A',a)}$ .

If no clause  $A' \leftarrow B'$ , appearing in the derivation prior to  $A \leftarrow B$ , satisfies  $A'\theta_G^{(A',a)} = A\theta_G^{(A,a)}$ , then by the definition of  $R^D$ ,  $(A \leftarrow B)\theta_G^{(A \leftarrow B,a)}$  is a clause in  $R^D$ . As in the case where  $A \leftarrow B$  is an instance of a clause in  $P$ , we conclude, by the induction hypothesis, that the atoms in  $B\theta_G^{(B,a)}$  can be derived from  $P \cup R^D$  and therefore  $A\theta_G^{(A,a)}$  can be derived from  $P \cup R^D$ . ■

**Lemma 5.3.** *Every goal  $G'$  which can be derived from  $(Q, G_Q) \cup (R^D, G_R)$  can be derived from  $(Q, G_Q) \cup (R, G_R)$ .*

*Proof.* In fact, by the definition of  $R^D$ , every clause in  $R^D$  is an instance of a clause in  $R$ . Therefore, due to the derivation rules (defined in the Appendix), every derivation of  $G'$  from  $(Q, G_Q) \cup (R^D, G_R)$  is also a derivation of  $G'$  from  $(Q, G_Q) \cup (R, G_R)$ . ■

**Proposition 5.3.** *For every set of semantic relations  $R$ ,  $\langle NFA; \cup; R \cup \{\simeq, \equiv\} \rangle$  is a testing sub-structure for  $\langle AFA; \cup; R \cup \{\simeq, \equiv\} \rangle$ .*

*Proof.* Let  $P, Q \in AFA$  be automata such that  $P \simeq Q$  but  $P \neq Q$ . Let  $R \in AFA$  be an automaton which distinguishes between  $P$  and  $Q$ . Without loss of generality, there is a word  $C$  such that  $C \in L(P \cup R)$  but  $C \notin L(Q \cup R)$ . From the following lemma, we conclude that there is a non-alternating automaton  $R' \in NFA$  which distinguishes between  $P$  and  $Q$ .

**Lemma 5.4.** *Let  $P, Q, R \in AFA$  and let  $C$  be a word such that  $C \in L(P \cup R)$  but  $C \notin L(Q \cup R)$ . There exists a non-alternating automaton  $R' \in NFA$  and a word  $C_1$  such that  $C_1 \in L(P \cup R')$  and  $C_1 \notin L(Q \cup R')$ .*

We prove the lemma by induction on the number of universal transitions, employed in the derivation of  $C$  from  $P \cup R$ . If no universal transitions are employed in this derivation, then  $C_1 = C$  and  $R'$ , which is the automaton contained in  $R$ , consisting only of the states and transitions employed in the derivation of  $C$ , satisfies the required conditions.

Otherwise, assume the lemma holds for every automaton  $R$  and a word  $C$  such that the derivation of  $C$  in  $P \cup R$  employs at most  $k$  universal transitions of  $R$ . We now show that the lemma holds also if  $C$  requires at most  $k + 1$  universal transitions of  $R$  when derived from  $P \cup R$ . Let  $t = (q, a, \{q_1, \dots, q_n\})$  be the first universal transition of  $R$ , employed in the derivation of  $C$  from  $P \cup R$  ( $q$  is a universal state of  $R$  in the shortest distance from  $q_0$ ). Let  $C'$  be the prefix obtained in the path from  $q_0$  to  $q$  in  $P \cup R$ .  $C'$  satisfies one of the following cases:

**$C'$  can be obtained in  $Q \cup R$  by reaching the state  $q$ .** If there is a suffix  $C''$  obtained by first applying the transition  $t$  to  $q$  in  $P \cup R$ , and this suffix cannot be obtained in  $Q \cup R$ , then we add a new letter  $a'$ , not appearing in  $Q$  and a triple  $(q, a', q')$ , where  $q'$  is the successor of  $q$  in the derivation of the suffix  $C''$ , to the transition relation of  $R$ . With this extended automaton  $R''$  we can get a derivation in  $P \cup R''$  for the word  $C''$  composed of  $C'$  as a prefix and  $C''$  as a suffix, with the first letter of  $C''$  replaced by the letter  $a'$ . This derivation employs at most  $k$  universal transitions of  $R''$  (since it can use the same transitions employed in the derivation of  $C$  from  $P \cup R$ , with the universal transition  $t$  replaced by the new added transition). In addition  $C'' \in L(P \cup R'')$  while  $C'' \notin L(Q \cup R'')$  (by the properties of  $q$  and since  $a'$  is a new letter). Therefore, by the induction hypothesis, there exists an automaton  $R'$  and a word  $C_1$  as required.

If every suffix obtained from  $q$  in  $P \cup R$ , beginning with  $t$ , can be obtained in  $Q \cup R$  as well, we define a new automaton  $R''$ , which is  $R$  extended with the non-universal transitions  $(q, a, q_i)$ ,  $1 \leq i \leq n$ , where  $t = (q, a, \{q_1, \dots, q_n\})$ .  $C$  can be derived in  $P \cup R''$  with at most  $k$  universal transitions of  $R''$ , since the universal transition  $t$  can be replaced by one of the new transitions.  $C \notin L(Q \cup R'')$  (since the new transitions already lead to acceptance in  $Q \cup R$  and hence cannot affect the distinction between  $P$  and  $Q$  using  $C$ ). Hence by the induction hypothesis, there exists  $R'$  and  $C_1$  as required.

**$C'$  can be obtained as a prefix in  $Q \cup R$ , but a state  $q'$  different from  $q$  is reached.** Let  $R''$  be the automaton obtained from  $R$  by adding a new state  $q''$  (not in the states of  $P$ ,  $Q$ , or  $R$ ) to the final states of  $R$  and the triple  $(q, a', q'')$  to the transition relation of  $R$ , where  $a'$  is a new letter, not appearing in  $Q$ . Let  $C'' \stackrel{\text{def}}{=} C' \cdot a'$ . Then,  $C'' \in L(P \cup R'')$  and the derivation of  $C''$  from  $P \cup R''$  employs at most  $k$  universal transition of  $R''$  (it can employ the new transition instead of  $t$ ). Moreover,  $C'' \notin L(Q \cup R'')$  since  $a'$  doesn't appear in  $Q$  and the prefix  $C'$  does not reach  $q$  in  $Q \cup R$ . Therefore, by the induction hypothesis, there exists  $R'$  and  $C_1$  as required.

**$C'$  cannot be obtained as a prefix in  $Q \cup R$ .** Let  $R''$  be the same automaton as  $R$  with the only exception that  $q$  is an accepting state. In  $P \cup R''$ ,  $C'$  has a derivation which employs at most  $k$  universal transitions of  $R''$  (it doesn't have to

employ the universal transition  $t$ ). In addition,  $C' \notin L(Q \cup R')$  since  $C'$  cannot be obtained in  $Q \cup R$ . By the induction hypothesis, there exists a non-alternating automaton and a separating word as required. ■

**Proposition 5.4.** *For every set of semantic relations  $R$ ,  $\langle DFA; \cup; R \cup \{\simeq, \equiv\} \rangle$  is a testing sub-structure for  $\langle NFA; \cup; R \cup \{\simeq, \equiv\} \rangle$ .*

*Proof.* Similar to the previous case, let  $P, Q, R \in NFA$  be automata and let  $C$  be a word such that  $C \in L(P \cup R)$  but  $C \notin L(Q \cup R)$ . From the following lemma, we conclude that there is a deterministic automaton  $R' \in DFA$  which distinguishes between  $P$  and  $Q$ .

**Lemma 5.5.** *Let  $P, Q, R \in NFA$  and let  $C$  be a word such that  $C \in L(P \cup R)$  but  $C \notin L(Q \cup R)$ . There exists a deterministic automaton  $R' \in DFA$  and a word  $C_1$  such that  $C_1 \in L(P \cup R')$  and  $C_1 \notin L(Q \cup R')$ .*

A pair  $(q, a)$  of a state and an input letter in a derivation is called *non-deterministic with respect to an automaton  $A$*  if there are at least two different states  $q_1, q_2$  in  $A$  such that both transitions  $(q, a, q_1)$  and  $(q, a, q_2)$  of  $A$  are employed in the derivation.

We prove the lemma by induction on the number of non-deterministic pairs with respect to  $R$ , in the derivation of  $C$  from  $P \cup R$ . If there isn't any non-deterministic pair with respect to  $R$  in this derivation, then  $C_1 = C$  and  $R'$ , which is the automaton contained in  $R$ , consisting only of the states and transitions employed in the derivation of  $C$ , satisfies the required conditions.

Otherwise, assume the lemma holds for every automaton  $R$  and a word  $C$  such that in the derivation of  $C$  in  $P \cup R$  there are at most  $k$  non-deterministic pairs with respect to  $R$ . We now show that the lemma holds also when the derivation of  $C$  from  $P \cup R$  contains at most  $k + 1$  such pairs. Let  $(q, a)$  be a non-deterministic pair with respect to  $R$  in the derivation of  $C$  from  $P \cup R$ . The derivation of  $C$  from  $P \cup R$  can be partitioned to segments as follows:

**An initial segment:** The part of derivation from  $q_0$  to the first occurrence of  $q$  to which an  $R$ -transition of the form  $(q, a, q')$  is applied. We denote by  $C_0$  the prefix of input letters obtained in this part of the derivation.

**Middle segments:** A non-empty part of derivation segments, between successive applications of  $R$ -transitions having the form  $(q, a, q')$ . We denote the sequences of input letters read in these segments by  $C_1, \dots, C_n$  ( $n \geq 1$  since  $(q, a)$  is a non-deterministic pair in the derivation of  $C$ ).

**A final segment:** The segment beginning with the last application of an  $R$ -transition

of the form  $(q, a, q')$  and terminating with an accepting state. We denote the suffix of input letters read by this segment by  $C_{n+1}$ .

If  $C_0$  can be obtained in  $Q \cup R$ , then since  $C \notin L(Q \cup R)$ , there is a minimal  $1 \leq i \leq n + 1$  such that  $C_0 \cdots C_{i-1}$  can be obtained in  $Q \cup R$ , while  $C_0 \cdots C_i$  cannot be obtained. Define  $R''$  to be the automaton accepted from  $R$  by adding new letters  $a_1, \dots, a_i$  not appearing in  $Q, P$  or  $R$ , and new triples  $(q, a_j, q_j)$ ,  $(1 \leq j \leq i)$ , to the transition relation of  $R$ , where  $(q, a, q_j)$  is the first transition employed in order to derive  $C_j$ . Then,

$$C' \stackrel{\text{def}}{=} \begin{cases} C_0 \cdot C'_1 \cdots C'_i \cdot C_{n+1} & \text{if } i \leq n, \\ C_0 \cdot C'_1 \cdots C'_i & \text{otherwise,} \end{cases}$$

where for every  $1 \leq j \leq i$   $C'_j$  is obtained from  $C_j$  by replacing its first  $a$  by  $a_j$ .  $C'$  has a derivation in  $P \cup R''$  in which the pairs  $(q, a)$  as well as  $(q, a_1), \dots, (q, a_i)$  are deterministic with respect to  $R''$ , and hence there are at most  $k$  non-deterministic pairs with respect to  $R''$  in this derivation. Furthermore,  $C' \notin L(Q \cup R'')$  since  $C_0 \cdots C_i$  cannot be obtained in  $Q \cup R$  and since the letters  $a_1, \dots, a_i$  are new. By the induction hypothesis, there exists a deterministic automaton and a separating word as required.

Otherwise,  $C_0$  cannot be obtained in  $Q \cup R$ . Let  $R''$  be the automaton accepted from  $R$  by changing  $q$  to be a final state.  $C_0$  has a derivation in  $P \cup R''$ , which is a sub-derivation of the derivation of  $C$  from  $P \cup R$ . This derivation does not include any  $R$ -transition of the form  $(q, a, q')$ . Therefore, this derivation includes at most  $k$  non-deterministic pairs with respect to  $R''$  (the pair  $(q, a)$  is no longer relevant). Furthermore,  $C_0 \notin L(Q \cup R'')$  since the prefix  $C_0$  cannot be obtained in  $P \cup R$ . Hence, by the induction hypothesis there is a deterministic automaton and a separating word as required. ■

## 5.2. EMBEDDING BETWEEN LOGIC PROGRAMS AND TURING MACHINES

We adopt the key ideas of the simulations presented in [22, 26] to conclude that there exist  $\mathcal{OF}$ -homomorphisms between the structures associated with logic programs and those associated with Turing machines. Based on the simulations presented in [22, 26], there exists an embedding of alternating Turing machines into logic programs. The restriction of this embedding to non-deterministic and deterministic Turing machines yields an embedding of non-deterministic Turing machines into iterative logic programs and of deterministic Turing machines into deterministic logic programs. The above embeddings are faithful with respect to the observable equivalence, as well as the *simple* relations. Similarly, there exists an embedding of logic programs into alternating Turing machines. Restricting this embedding to iterative logic programs, we get an embedding of iterative logic

programs into non-deterministic Turing machines. Both embeddings are faithful with respect to the observable equivalence relation.

### 5.2.1. Embedding (initialized) logic programs in Turing machines

#### Theorem 5.2. $LP \preceq_{OF} ATM$

*Proof.* We introduce an embedding of logic programs into Turing machines. The embedding is based on the simulation presented in [26]. It is slightly revised in order to handle *initialized* programs.

Given an initialized logic program  $(P, G)$ , we define  $M = (P, G)_\varepsilon$  to be an alternating Turing machine described as follows. The program  $P$  and the set of exported predicates  $G$  are stored in the finite control of  $M$ . In the initial state  $q_0$ , a ground goal  $A$  is written on the tape of  $M$ . From its initial state,  $M$  proceeds as follows. If  $G \neq \emptyset$  the machine checks that the predicate of  $A$  is included in  $G$  and if so, it reaches a state called the *reduction state*. From the reduction state, using existential branching, it chooses a clause  $A' \leftarrow B_1, \dots, B_k$  in  $P$  and writes on its tape a ground substitution  $\theta$ . It then computes  $A'\theta$ , verifies that  $A = A'\theta$ , and erases everything from the tape except  $\theta$ . Then, using universal branching, it chooses  $B_i$  for some  $i$ , applies  $\theta$  to  $B_i$ , erases everything from the tape except  $B_i\theta$  and returns to the reduction state. As a result, the final states of  $M$  are the universal states corresponding to unit clauses.

The embedding definition (especially the unambiguous encoding of  $P$  in the machine's finite control) ensures that the states simulating the reduction of different clauses are disjoint. Therefore, the embedding defined above is a homomorphism. From the discussion in [26] and due to the modifications added to the above embedding,  $M$  accepts  $A$  if and only if  $A$  can be derived from  $(P, G)$ . Therefore the above embedding is faithful with respect to  $\simeq$ . ■

By restricting the above embedding to the sub-structure  $ILP$  we get the following theorem.

#### Theorem 5.3. $ILP \preceq_{OF} NTM$

*Proof.* The restriction of the embedding described in Theorem 5.2 to iterative logic programs constitutes an  $OF$ -homomorphism of  $ILP$  into  $NTM$ . ■

### 5.2.2. Embedding Turing machines in (initialized) logic programs

Based on the simulation described in [26], we present an embedding of alternating Turing machines in logic programs. The restriction of this embedding to non-deterministic and deterministic Turing machines yields an embedding of non-

deterministic Turing machines in iterative logic programs, and of deterministic Turing machines in deterministic logic programs.

**Theorem 5.4.**  $ATM \preceq_{\mathcal{O}\mathcal{F}} \mathcal{L}\mathcal{P}$

*Proof.* Given an alternating Turing machine  $M$ , we define  $M\varepsilon = (P, G)$  to be the following initialized logic program. The clauses in  $P$  consist of atoms of the form  $q(L, R)$ , where  $q \in Q$  is a state of  $M$ , and  $L, R$  are lists of  $\Sigma$ -elements ( $\Sigma$  is the alphabet of  $M$ ) describing the used part of the tape to the left and to the right of the machine head. Thus an atom  $q(L, R)$  corresponds to an  $M$ -configuration  $\langle L, q, R \rangle$ . The meaning of the predicate  $q(L, R)$  is that the corresponding  $M$ -configuration  $\langle L, q, R \rangle$  leads to acceptance.

For every state  $q$  and symbol  $\sigma$  such that  $(q, \sigma, \{q_1, \dots, q_n\}, \sigma', LR)$  is an  $M$ -transition, we include in  $P$  clauses of the form

$$q(L, R) \leftarrow q_1(L_1, R_1), \dots, q_n(L_n, R_n).$$

Several clauses are required in order to describe the various possible relations between the old and the new configurations, following the employed transition (such as moving beyond the ends of the used part of the tape, an empty tape *etc.*). A detailed description of these clauses can be found in [26]. Note that if  $q$  is existential, we get iterative clauses. Otherwise we get definite clauses. If  $q$  is final we get a unit clause.

We define the set of exported predicates to be  $G = \{q_0\}$  only if  $M$  has a transition of the form  $(q_0, \sigma, Q', \sigma', LR)$ ; ( $G = \emptyset$  otherwise).

Due to the modified definition employed for the transition relation in  $ATM$ , it is easy to see that the above embedding is indeed a homomorphism. Following the discussion in [26], it is faithful with respect to the observable equivalence relation. Furthermore, in our representation of Turing machines, redundant states (non-final states which are not employed by any transition) are not taken into consideration (*i.e.* machines with redundant states are identified with the machines obtained by omitting the redundant states). Using such a representation, the above embedding is faithful with respect to the *simple* relation as well. ■

By restricting the above embedding to non-deterministic and deterministic Turing machines, we get the following.

**Theorem 5.5.**  $NTM \preceq_{\mathcal{O}\mathcal{F}} \mathcal{I}\mathcal{L}\mathcal{P}$

*Proof.* We restrict the embedding described in Theorem 5.4 to non-deterministic Turing machines. Since, by definition, a transition whose source state is purely non-deterministic (non-alternating) is translated to an iterative clause, the restricted embedding maps non-deterministic Turing machines to iterative logic programs.

Using the same arguments as in Theorem 5.4, the resulting embedding is faithful with respect to the observable equivalence as well as the *simple* relations. ■

**Theorem 5.6.**  $DTM \preceq_{\mathcal{OF}} DCP$

*Proof.* We restrict the embedding described in Theorem 5.4 to deterministic Turing machines. By carefully examining a logic program which is the image of *any* deterministic Turing machine under the embedding defined in Theorem 5.4 (and further detailed in [26]), we conclude the following:

- For every clause, all clause variables are head variables.
- Since the source machine is deterministic, clause heads are pairwise non-unifiable. (For every combination of a state and a symbol read by the scanning head there corresponds at most a unique transition, and clauses describing the same transition have non-unifiable heads.)

Therefore the restricted embedding maps deterministic Turing machines into deterministic logic programs. To complete the proof, it is easy to see that as a restricted embedding, it is also homomorphic and faithful with respect to the observable equivalence relation, as well as the *simple* relation. ■

### 5.3. EMBEDDING FINITE AUTOMATA IN TURING MACHINES

We present an  $\mathcal{OF}$ -homomorphism of alternating finite automata into alternating Turing machines. By restricting this embedding to non-deterministic finite automata (without alternation) and to deterministic finite automata, we get  $\mathcal{OF}$ -homomorphisms of non-deterministic finite automata into non-deterministic Turing machines and of deterministic finite automata into deterministic Turing machines. The homomorphisms presented in this section preserve the *simple* relation as well.

**Theorem 5.7.**  $AFA \preceq_{\mathcal{OF}} ATM$

*Proof.* Given an alternating finite automaton  $(\delta, U, F)$ , we extend the ternary transition relation  $\delta$  to a 5-ary transition relation  $\delta'$  as follows. For each  $(q, a, q') \in \delta$  such that  $q' \notin F$ , we define  $\delta'$  to include the quintet  $(q, a, q', a, R)$ . For  $(q, a, q_F) \in \delta$  with  $q_F \in F$ , we define  $\delta'$  to include the quintet  $(q, a, q^*, a, R)$  (where  $q^*$  is a new state, not appearing in the automaton transition relation) as well as the quintet  $(q^*, bl, q_F, bl, R)$ . (When viewing  $\delta'$  as a transition relation for a Turing machine, it means that upon reading an input character, the Turing machine changes its state as the finite automaton does, its head always moves to the right, and it never changes the contents of the tape. When the automaton reaches a final state the Turing machine checks that the input word is terminated.) We now define the Turing machine  $(\delta', U, F)$  to be the image of the automaton  $(\delta, U, F)$ . This

constitutes an embedding  $\varepsilon$  of  $\mathcal{AFA}$  in  $\mathcal{ATM}$ . This embedding is homomorphic since it maps the automaton components to themselves, and union is defined analogously in Turing machines and in finite automata. It is faithful with respect to the observable equivalence relation since it preserves the accepted language. Furthermore, it is easy to ensure that this embedding is also faithful with respect to the *simple* relation. ■

**Theorem 5.8**

1.  $NFA \preceq_{OF} NTM$
2.  $DFA \preceq_{OF} DTM$

*Proof.* We use the restrictions of the embedding presented in Theorem 5.7. Obviously, the above embedding maps non-alternating automata to non-alternating machines, and deterministic automata to deterministic machines. Furthermore, the homomorphism property and the property of being faithful with respect to the observational equivalence relation and the *simple* relation hold also in the restricted embedding. This completes the proof of the theorem. ■

**6. Conclusions and future work**

The results presented in this paper demonstrate the generality of the comparison method introduced in [28] and its wide possible range of use. This research may be completed either by proving that the embeddings introduced by dashed arrows in Figure 1 preserve the fully-abstract congruence and the *simple* relations as well, or by strengthening the negative results not to require these relations to be preserved. By establishing one of the above proofs, all the relations in this paper will concern the same type of associated structures and hence employ the same degree of comparison. In addition, there are still missing arrow-heads (or separation lines) in Figure 1, which correspond to open problems about the relation between languages. Another direction for completing the picture is to investigate the relationship between alternating machines and *concurrent* machines [10, 15, 17], and to extend the separation results to other deterministic, non-deterministic and alternating models such as the fixpoint logics described in [1].

**Appendix: Bottom-up derivation rules for logic programs**

We employ the following rules in order to derive a ground goal  $G$  from a logic program  $P$ .

**MP: Modus ponens**

$$\frac{A \leftarrow B, B}{A}$$

**CI: Conjunction introduction**

$$\frac{A, B}{A \& B}$$

**Universal quantification**

$$\frac{\forall_X H \leftarrow B}{(H \leftarrow B)\theta} \text{ where } \theta \text{ is a substitution and } X \subseteq \text{dom}(\theta)$$

A *bottom-up derivation* of  $P$  is a sequence of logical expressions, each of which is either the atom *true*, or a quantified clause of  $P$  or an expression which is derived from previous expressions, using one of the above derivation rules.

A goal  $G$  is derived from the logic program  $P$  if there is a (bottom-up) derivation of  $P$ , where  $G$  is one of the expression in that derivation.

**References**

- [1] S. Abiteboul, M.Y. Vardi and V. Vianu, Fixpoints logics, relational machines, and computational complexity, *Proc. 7th Structure in Complexity Theory Conference*, 1992, pp. 156–168.
- [2] F. Afrati and S.S. Cosmadakis, Expressiveness of restricted recursive queries, *Proc. of the 21st ACM Symp. on Theory of Computing*, 1989, pp. 113–126.
- [3] L. Bougé, On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes, *Acta Informatica* 25 (1988) 179–201.
- [4] S. Brown, D. Gries and T. Szymanski, Program schemes with pushdown stores, *SIAM J. Comput.* 1 (1972) 242–268.
- [5] A.K. Chandra, D.C. Kozen and L.J. Stockmeyer, Alternation, *J. ACM* 28(1) (1981) 114–133.
- [6] A.K. Chandra and L.J. Stockmeyer, Alternation, *Proc. of the 17th IEEE Symp. on Foundation of Computer Science*, 1976, pp. 98–108.
- [7] R.L. Constable and D. Gries, On classes of program schemata, *SIAM J. Comput.* 1 (1972) 66–118.
- [8] F.S. de Boer and C. Palamidessi, Concurrent logic programming: asynchronism and language comparison, *Proc. of 1990 North American Conference on Logic Programming*, S. Debray and M. Hermenegildo (eds.) (MIT Press, 1990) pp. 175–194.
- [9] F.S. de Boer and C. Palamidessi, Embedding as a tool for language comparison: On the CSP hierarchy, *Proc. of the 2nd International Conference on Concurrency Theory (CONCUR'91)*, J.C.M. Baeten and J.F. Groote (eds.), LNCS 527 (Springer-Verlag, 1991) pp. 127–141.
- [10] D. Drusinsky and D. Harel, On the power of bounded concurrency I: The finite automata level, to appear in *JACM*. Earlier version appeared as: "On the power of cooperative concurrency", *Proc. Concurrency '88*, Lecture Notes in Computer Science, Vol. 335 (Springer-Verlag, New York, 1988) pp. 74–103.
- [11] M. Felleisen, On the expressive power of programming languages, *Proc. ESOP'90*, N. Jones (ed.), LNCS 432 (Springer-Verlag, 1990) pp. 134–151. A full version appeared in *Science of Computer Programming*, Vol. 17 (Elsevier, 1991) pp. 35–75.
- [12] M.J. Fischer and R.A. Ladner, Propositional dynamic logic of regular programs, *J. Comp. and Syst. Sci.* 18 (1979) 194–211.
- [13] H. Gaifman and E. Shapiro, Fully abstract compositional semantics for logic programs, *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989, pp. 134–142.

- [14] H. Gaifman and E. Shapiro, Proof theory and semantics of logic programs, *Proceedings of the 4th Annual Symposium on Logic in Computer Science*, Pacific Grove, California, 1989, pp. 50–62.
- [15] D. Harel, A thesis for bounded concurrency, *Proc. of the 14th Symp. on Math. Found. of Comput. Sci.*, Lecture Notes in Computer Science, Vol. 379 (Springer-Verlag, New York, 1989) pp. 35–48.
- [16] M. Hennessy, *Algebraic Theory of Processes* (MIT Press, 1988).
- [17] T. Hirst and D. Harel, On the power of bounded concurrency II: The pushdown automata level, *Proc. CAAP '90, Trees in Algebra and Programming*, Lecture Notes in Computer Science, Vol. 431 (Springer-Verlag, 1990) pp. 1–17.
- [18] P.J. Landin, The next 700 programming languages, *Comm. ACM* 9(3) (1966) 157–166.
- [19] A.R. Meyer and M.J. Fischer, Economy of description by automata, grammars, and formal systems, *Proc. of the 12th IEEE Symp. on Switching and Automata Theory*, 1971, pp. 188–191.
- [20] D. Kozen, On parallelism in Turing machines, *Proc. of the 17th IEEE Symp. on Foundation of Computer Science*, 1976, pp. 89–97.
- [21] J. Mitchell, On abstraction and the expressive power of programming languages, *Proc. of the International Conference on Theoretical Aspects of Computer Science*, LNCS (Springer, Sendai, Japan, 1991) pp. 290–310.
- [22] Y. Okabe and S. Yajima, Parallel computational complexity of logic programs and alternating Turing machines, *Proc. of the International Conference on Fifth Generation Computer Systems (ICOT, 1988)* pp. 356–363.
- [23] M.S. Paterson and C.E. Hewitt, Comparative schematology, *Record of Project, MAC Conference on Concurrent Systems and Parallel Computation* (ACM, 1970) pp. 119–127.
- [24] J.C. Reynolds, The essence of ALGOL, in *Algorithmic Languages*, de Bakker and van Vilet (eds.) (North Holland, Amsterdam, 1981) pp. 345–372.
- [25] J.G. Riecke, Fully abstract translations between functional languages (Preliminary Report), *Proc. ACM POPL*, 1991.
- [26] E. Shapiro, Alternation and the computational complexity of logic programs, *J. Logic Programming* 1 (1984) 19–33.
- [27] E. Shapiro, Embeddings among concurrent programming languages, *Proc. of the 3rd International Conference on Concurrency Theory (CONCUR'92)*, W.R. Cleaveland (ed.), LNCS 630 (Springer-Verlag, 1992) pp. 486–503.
- [28] E. Shapiro, Separating concurrent languages with categories of language embeddings, *Proc. STOC'91* (ACM, 1991) pp. 198–208.
- [29] G.L. Steele, Jr. and G.J. Sussman, Lambda — The ultimate imperative, MIT AI Memo 353 (1976).
- [30] F.W. Vaandrager, On the relationship between process algebra and input/output automata, *Proc. LICS'91* (IEEE, 1991) pp. 387–398.