Hardware Verification, Boolean Logic Programming, Boolean Functional Programming

Enrico Tronci[1, 2]

Dip. Matematica Pura ed Applicata, Univeristà di L'Aquila, Coppito, 67100 L'Aquila, Italy

Abstract

One of the main obstacles to automatic verification of Finite State Systems (FSSs) is state explosion. In this respect automatic verification of an FSS M using Model Checking and Binary Decision Diagrams (BDDs) has an intrinsic limitation: no automatic global optimization of the verification task is possible until a BDD representation for M is generated. This is because systems and specifications are defined using different languages. To perform global optimization before generating a BDD representation for M we propose to use the same language to define systems and specifications.

We show that First Order Logic on a Boolean Domain yields an efficient functional programming language that can be used to represent, specify and automatically verify FSSs. E.g. on a SUN Sparc Station 2 we were able to automatically verify a 64 bit commercial multiplier.

Key words: Hardware Verification, Model Checking, Boolean Functional Programming, Boolean Logic Programming, Boolean First Order Logic, μ-calculus, Binary Decision Diagrams.

1. Introduction

One of the main obstacles to automatic verification of Finite State Systems (FSSs) is state explosion since it can quickly fill up a computer memory. Nevertheless FSSs of considerable size have been automatically verified using μ-calculus Model Checking on a Boolean Domain (e.g. see [2]) and efficient canonical representations for Boolean Functions (namely: Binary Decision Diagrams, BDDs, see [3]). Model Checking (MC), however, has an intrinsic limitation: it does not allow automatic global optimization of the verification task *until* a (BDD) representation for the system to be verified is generated. Removing such limitation will allow us to enlarge the class of FSSs automatically verifiable. The following will clarify the matter.

In our context an MC problem can be seen as a pair (M, φ), where M (the model) is a finite list of boolean functions and φ (the specification) is a map assigning a boolean function eval(M, φ) to M (i.e to the list of boolean functions in M). Moreover answer(M, φ) = **if** eval(M, φ) is identically equal to true **then** true **else** false. For a large class of properties (see [2], [9]) automatic verification of FSSs comes down to compute answer(M, φ). Efficient canonical representations for boolean functions are crucial to carry out such computation. BDDs have been very successful in this respect.

A global optimization is a transformation taking an MC problem (M, φ) and returning an (hopefully) *easier* MC problem (M', φ') s.t. answer(M, φ) = answer(M', φ'). E. g. in [2] sec. 5 and [11] are optimization techniques in which φ (but not M) is modified to improve fixpoint computation performances. All MC optimization techniques that we know of act only on φ. However to avoid state explosion when dealing with combinatorial circuits we need to modify M and φ. This is because BDDs are a canonical form for boolean functions. To the best of our knowledge no automatic global (i.e. acting on both M and φ) optimization technique has been presented in the literature.

Automatic global optimization in an MC setting is difficult because model M and specification φ are defined using different languages. E.g. M can be defined using Hardware Description Languages, Process Algebras, etc., whereas specification φ is usually defined using logic (e.g. μ-calculus or a temporal logic). For this reason it is hard to compare M and φ *until* BDDs representing M are generated. This can be far too late to avoid running out of memory. To be effective global optimization should take place *before* a BDD representation for M is generated. To this end we need to define M and φ in the same language. If such a language, say L, is available then given descriptions for M and φ (in any two languages) we can translate them into L and then carry out verification in L.

We show that *Logic* can be used as a common

1. email: tronci@smaq20.univaq.it.

2. This work has been partially supported by MURST funds.

language for systems and system specifications. This idea is already in, e.g., TLA ([14]), HOL ([10]), Coq ([8]), Nuprl ([7]). However such approaches rely on very powerful logics, thus they do not yield automatic verifiers. To get an efficient logic based automatic verifier we need to restrict ourselves to a language L satisfying the following contrasting requirements:

- To each formula in L we can associate a model M and efficiently compute a representation for M.
- L is large enough to express interesting specifications.

We show that *Boolean First Order Logic* (BFOL) can be used to represent systems and system specifications. In particular we show that BFOL can be used as a functional programming language which, in turn, yields an efficient logic based automatic verifier for FSSs. To stress the logic-functional nature of our BFOL based functional programming language we call it BFP (*Boolean Functional Programming*). Syntactically BFP is a *Logic Programming Language on a Boolean Domain*. However we do not use resolution, instead we compute on BFOL models using BDDs. Thus computationally BFP behaves as a *Functional Programming Language on a Boolean Domain* with BDDs playing the same role as λ-calculus for *traditional* functional programming.

The main results in this paper are:

- BFP is as expressive (3.7) and at least as efficient (3.9) as μ-calculus MC via BDDs. Moreover all algorithms developed in an MC setting (e.g. [2] sec. 5) can also be used with BFP. We point out that a BDD based (imperative) programming language already exists: Ever [12]. However, as far as we can tell, Ever does not support automatic global optimization.
- BFP allows automatic global optimization. This is performed by means of *program transformations*. This allows automatic verification of systems that are out of reach for Model Checking and BDDs alone. We present (4.2, 4.3, 5.4) a program transformation based on a syntactic and a semantic (via BDDs) analysis of the verification task. Simultaneous use of syntax and semantics makes BFP more efficient than MC. Our program transformation is effective also on combinatorial circuits. Note that no MC optimization acts on them.
- BFP is useful in practice. We are implementing (in C) a compiler for BFP. Using BFP on a SUN Sparc Station 2 we were able to automatically verify the correctness of a 64 bit commercial multiplier (section 6). A task out of reach for MC and BDDs alone.

BFP shows that using *simultaneously* Logic and efficient representations (BDDs) for Models we can improve computational performances of automatic verifiers for finite state systems. Moreover BFP provides an efficient BDD based logic-functional programming language to execute BFOL specifications.

The rest of this paper is organized as follows. In *section* 2 we review and adapt standard definitions from Logic Programming. In *section* 3 we define a class of programs suitable for automatic verification of FSSs. In *section* 4 we define a program transformation performing global optimization. In *section* 5 we give a reduction strategy allowing practical use of our program transformation. In *section* 6 we report on experimental results on automatic verification of a commercial multiplier.

2. Basic Definitions

In this section we review standard definitions from Logic Programming (see, e.g., [1], [15]) and adapt them to our case: *Boolean Functional Programming* (BFP). *Boolean First Order Logic* (BFOL) is simply first order logic on the boolean domain {0, 1}. Symbol $\equiv$ denotes syntactic equality between strings. An identifier is just an alphanumeric string.

2.0. Definition. • An *alphabet* consists of: *Propositional Constants*: $\underline{0}$, $\underline{1}$; A countable set of identifiers called *Variables* (denoted by $x_0, x_1, \ldots$); A countable set of identifiers called *Predicate Symbols* (denoted by $p_0, p_1, \ldots$); *Connectives*: $\neg$, $\vee$, $\wedge$, $\rightarrow$, $=$, $\oplus$; *Quantifiers*: $\exists$, $\forall$; *Punctuation Symbols*: "(", ")", ",". We assume that the set of variables is infinite, disjoint by the set of predicate symbols and fixed once and for all. Thus an alphabet is univocally defined by its set of predicate symbols (which may be empty). To each predicate symbol p is associated an integer n ($n \geq 0$) called the arity of p (and p is said n-ary). Note that we have no function symbols.

- A *term* is a propositional constant or a variable. *Formulas* are defined as follows: any term is a formula (this is because we are working on Booleans), if p is an n-ary predicate symbol and $t_1, \ldots t_n$ are terms then $p(t_1, \ldots t_n)$ is a formula (called *atomic formula* or *atom*), if F and G are formulas then $(\neg F)$, $(F \vee G)$, $(F \wedge G)$, $(F \rightarrow G)$, $(F = G)$, $(F \oplus G)$ are formulas, if F is a formula and x is a variable then $(\exists x F)$, $(\forall x F)$ are formulas. As a syntactic sugar we suppress parenthesis as usual.
- A variable x occurs *free* in a formula F if x is not in the scope of a $\forall$ x or $\exists$ x; x occurs bound otherwise. FV(F) is the set of free variables in formula F.
- Formula F[x := t] is obtained from formula F by replacing all free occurrences of x in F with term t.
- Atom A *occurs* positively in A. If A *occurs* positively (negatively) in a formula F then A occurs positively (negatively) in: $(\exists x F)$, $(\forall x F)$, $(G \rightarrow F)$, $(F \text{ op } G)$, $(G \text{ op } F)$, where op is $\vee$, $\wedge$, $=$, $\oplus$. If A *occurs*

positively (negatively) in F then A occurs negatively (positively) in: ($\neg$ F), (F $\rightarrow$ G), (F op G), (G op F), where op is =, $\oplus$.

We use the usual semantics for first order languages. However our universe will always be the set Boole = {0, 1} of boolean values. Boolean value 0 stands for false and boolean value 1 stands for true. If b denotes a boolean value then we will often write b instead of $\underline{b}$. Thus symbols 0, 1 are overloaded since they can denote propositional constants, boolean values or *just* integers. However the formal context will always make clear the intended meaning. In the following it is always assumed that an alphabet is given.

2.1. Definition. • An *interpretation* I on a given alphabet is a subset of the set $\{p(t_1, \ldots t_n) \mid$ p is an n-ary predicate symbol in the given alphabet and $t_1, \ldots t_n$ are propositional constants$\}$. If G is a set of predicate symbols we define $I(G) = \{p(t_1, \ldots t_n) \mid p(t_1, \ldots t_n) \in I$ and $p \in G\}$. We write I(p) for I({p}). Let p be an n-ary predicate symbol and $v_1, \ldots v_n$ be boolean values. We define the boolean value $I(p)(v_1, \ldots v_n)$ as follows: $I(p)(v_1, \ldots v_n)$ = **if** $(p(\underline{v_1}, \ldots \underline{v_n}) \in I)$ **then** 1 **else** 0. Thus I(p) can also be regarded as an n-ary boolean function. Note that if n = 0 then symbol I(p) may also denote a boolean value. The mathematical context will always make clear the intended reading for I(p).

• A *state* is a map σ assigning a boolean value $\sigma(x)$ to each variable x. If σ is a state and d is a boolean value then $\sigma[x := d]$ is a state s.t.: $\sigma[x := d](y)$ = (**if** $(y \equiv x)$ **then** d **else** $\sigma(y)$). An *environment* is a map $[I, \sigma]$ assigning boolean values to terms and formulas as in Fig.2.0.

• Let P, S be set of formulas and I be an interpretation. We say that I is a *model* for S (notation: $I \models S$) iff for each formula F in S and for each state σ we have $[I, \sigma](F) = 1$. We say that S is a *logical consequence* of P (notation $P \models S$) iff for each interpretation I we have: if $I \models P$ then $I \models S$. If S = {F} then we write: $I \models F$, $P \models F$ for, respectively, $I \models \{F\}$, $P \models \{F\}$.

• A Model Checking (MC) problem is a pair (I, F), where I is an interpretation and F is a formula. Answer_MC is a function from Model Checking problems to Boole s.t.: Answer_MC(I, F) = 1 iff $I \models F$.

2.2. Definition. • A *program statement* is a formula of the form $p(x_1, \ldots x_n) = F$, where p is a predicate symbol and F is a formula s.t. $FV(F) \subseteq \{x_1, \ldots x_n\}$. Formulas $p(x_1, \ldots x_n)$ and F are called, respectively, the *head* and the *body* of the statement.

• A *program* is a finite nonempty set P of program statements s.t. for each predicate symbol p occurring in a program statement in P there is exactly one program statement in P with p occurring in the head. Thus a program is a system of *Boolean Functional Equations*. Note that we call program what in logic programming is usually called the *completion* of a logic program (see, e.g., [15] sec. 17, [1]).

• Let P be a program. A predicate symbol p is *in* P iff p occurs in a statement in P. The set Alph(P) of predicate symbols in P defines the alphabet of P. The *definition* of p in P is the (unique) program statement in P in which p occurs in the head. We denote with size(P) the number of symbols in P. A *model* for (or a *solution* to) P is an interpretation I s.t. $I \models P$.

Fig.2.0. $[I, \sigma](\underline{0}) = 0$, $[I, \sigma](\underline{1}) = 1$, $[I, \sigma](x) = \sigma(x)$,
$[I, \sigma](p(t_1, \ldots t_n)) = I(p)([I, \sigma](t_1), \ldots [I, \sigma](t_n))$,
$[I, \sigma](\neg F)$ = **if** $[I, \sigma](F)$ **then** 0 **else** 1,
$[I, \sigma](F \vee G)$ = **if** $[I, \sigma](F)$ **then** 1 **else** $[I, \sigma](G)$,
$[I, \sigma](F \wedge G)$ = **if** $[I, \sigma](F)$ **then** $[I, \sigma](G)$ **else** 0,
$[I, \sigma](F \rightarrow G)$ = **if** $[I, \sigma](F)$ **then** $[I, \sigma](G)$ **else** 1,
$[I, \sigma](F = G)$ =
  if $[I, \sigma](F)$ **then** $[I, \sigma](G)$ **else** $[I, \sigma](\neg G)$,
$[I, \sigma](F \oplus G)$ =
  if $[I, \sigma](F)$ **then** $[I, \sigma](\neg G)$ **else** $[I, \sigma](G)$,
$[I, \sigma](\exists x F)$ =
  if $[I, \sigma[x := 0]](F)$ **then** 1 **else** $[I, \sigma[x := 1]](F)$,
$[I, \sigma](\forall x F)$ =
  if $[I, \sigma[x := 0]](F)$ **then** $[I, \sigma[x := 1]](F)$ **else** 0.

We will need to carry out computations on BFOL interpretations. Thus we need to efficiently represent them. We will always deal with alphabets with a finite number of predicate symbols. Thus we can represent interpretations as finite lists of boolean functions. Boolean functions, in turn, can be represented using Binary Decision Diagrams (BDDs) (see [3] for details). BDDs are an efficient canonical representation for Boolean Functions. I.e. for each boolean function f there is (up to f argument ordering) exactly one BDD, bdd(f), representing f. In the following we assume that for each predicate symbol p (in the given alphabet) an ordering on its arguments is given. Thus, given an interpretation I, bdd(I(p)) is univocally determined. Moreover size_bdd(G) denotes the number of vertices in BDD G. We heavily rely on BDDs. However in the following they can be replaced by any efficient canonical representation for boolean functions.

In the following we refer to μ-calculus on a Boolean domain as defined in [2], with μ-calculus interpretations

defined as in 2.1. We will liberally use C-like pseudo-code for our algorithms.

In the following we omit proofs because of lack of space.

3. Regular Programs

Regular programs (3.1) are obtained adapting to BFOL μ-calculus formal monotonicity condition (e.g. see [2]). In 3.7 we show that regular programs are as expressive as μ-calculus on a boolean domain. In 3.9 we show that regular programs are at least as efficient as μ-calculus MC via BDDs. Note that CTL formulas can be expressed as μ-calculus terms (see [2]).

In 3.0 - 3.1 we define regular programs.

3.0. Definition. Let P be a program.

- We define the sets *pos_calls*(P, p), *neg_calls*(P, p) as in Fig.3.0. Intuitively these sets represent, respectively, the set of predicate symbols *involved* positively (negatively) in the definition of p in P.
- Let p, q be predicate symbols in P. We say that p *calls* q in P iff $q \in$ (*pos_calls*(P, p) $\cup$ *neg_calls*(P, p)). Intuitively p calls q in P iff q is *involved* in the definition of p in P.
- Let p be a predicate symbol in P and Q be a set of predicate symbols in P. We say that p *calls* Q in P iff there exists $q \in Q$ s.t. p calls q.

3.1. Definition. • A program P is said to be *regular* iff for all predicate symbols p in P, $p \notin$ *neg_calls*(P, p). Intuitively P is regular iff for each predicate symbol p in P all occurrences of p *involved* in the definition of p in P fall under an even number of negations.

Note that (the completion of) a stratified logic program (see [15] sec. 17) is a regular program. However it is not the case that every regular program is the completion of a stratified logic program. E.g. P = {p = (p $\vee \neg$q), q = (q $\vee \neg$p)} is a regular program, but it is not the completion of any stratified logic program.

In general a regular program does not have a unique solution. E.g. P as above has 3 solutions, namely: I_1 = {p}, I_2 = {q}, I_3 = {p, q}. However we want to use regular programs to univocally define models. Thus we need to associate to a program P a unique solution. This can be done in two ways: giving an algorithm that computes a solution to P or giving a property that univocally characterizes a solution to P. The second approach has many technical advantages. In particular it allows an easy comparison between different algorithms designed to find the same solution to a program P. We follow the second approach. This is done in 3.2 - 3.3.

Fig.3.0. To each predicate symbol p in a program P we associate sets *neg_calls*(P, p), *pos_calls*(P, p) as defined by *def_calls*.

```
void def_calls(program P) {
    for each predicate p in P do {
        neg_calls(P, p) = empty_set;
        pos_calls(P, p) = empty_set; } not_end = 1;
    while (not_end) { not_end = 0;
        for each statement p(t1, ... tn) = H in P do {
            for all predicates q occurring in an atom
            occurring positively in H do {
                temp = pos_calls(P, p) ∪ {q} ∪
                    pos_calls(P, q);
                if (temp ≠ pos_calls(P, p)) {not_end =
                    1; pos_calls(P, p) = temp; }
                temp = neg_calls(P, p) ∪
                    neg_calls(P, q);
                if (temp ≠ neg_calls(P, p)) {not_end =
                    1; neg_calls(P, p) = temp; }}
            for all predicates q occurring in an atom
            occurring negatively in H do {
                temp = neg_calls(P, p) ∪ {q} ∪
                    pos_calls(P, q);
                if (temp ≠ neg_calls(P, p)) {not_end =
                    1; neg_calls(P, p) = temp; }
                temp = pos_calls(P, p) ∪
                    neg_calls(P, q);
                if (temp ≠ pos_calls(P, p)) {not_end =
                    1; pos_calls(P, p) = temp; }}}}}
```

3.2. Definition. • From now on we assume that a linear order on the set of predicate symbols in the given alphabet is defined. We denote with ≤ such linear order. We write a < b for (a ≤ b and a ≠ b). In the following we assume that ≤ is just the lexicographic order on predicate symbols.

• A module is a nonempty subset of a program. More precisely a *module* is a finite nonempty set G of program statements s.t. for each predicate symbol p occurring in a program statement in G there is *at most* one program statement in G with p occurring in the head.

• Let G be a module. We denote with export(G) the set of predicate symbols defined in G. I.e. export(G) = {p | p is a predicate symbol occurring in the head of a program statement in G}. Let I be an interpretation. We extend 2.1 by writing I(G) for I(export(G)). We define the binary relation $\sqsubseteq_G$ on interpretations as follows:

If Card(G) = 1 then: $I \sqsubseteq_G I'$ iff $I(G) \subseteq I'(G)$;

If $G = \{p_1(x_1, \ldots x_{a(1)}) = F_1, \ldots p_n(x_1, \ldots$

$x_{a(n)}) = F_n\}$, $n > 1$, $p_1 < \ldots < p_n$ and $H = G - \{p_1(x_1, \ldots x_{a(1)}) = F_1\}$ then: $I \sqsubseteq_G I'$ iff $(I(p_1) \subseteq I'(p_1)$ and $(I(p_1) = I'(p_1)$ implies $I \sqsubseteq_H I'))$.

• Let P be a program, p be a predicate symbol in P and G be a set of program statements in P. We denote with def(P, p) the definition of p in P. I.e. def(P, p) is the unique statement in P in which p occurs in the head. We define: cluster(P, p) = {def(P, p)} ∪ {def(P, q) | (p calls q in P) and (q calls p in P)}, clusters(P) = {cluster(P, p) | p is in P}, base(P, G) = {p | (p is a predicate in P) and (p ∉ export(G)) and (there exists g ∈ export(G) s.t. g calls p in P)}.

• Let P be a program. A *standard solution* to P is an interpretation I s.t.: $I \models P$ and I = I(Alph(P)) and for all G ∈ clusters(P), for all interpretations I', if I(base(P, G)) ∪ I'(G) ⊨ G then $I \sqsubseteq_G I'$.

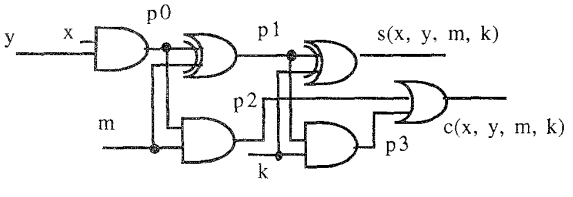
3.3. Theorem. Let P be a regular program. Then P has exactly one standard solution.

Proof. (*Sketch*). We give a constructive proof exploiting Tarsky-Knaster fixpoint theorem. See appendix A for more details.

3.4. Example. • Let P = {q = ¬p, p = p}. P is a regular program (as well as the completion of a stratified logic program). The standard solution to P is I = {q}.

• Let P = {p = (p ∨ ¬q), q = (q ∨ ¬p)}. P is a regular program. The standard solution to P is I = {q}.

Fig.3.1. P0 = Cell_I ∪ Cell_S,
P = P0 ∪ {g(x, y, m, k) = ((s(x, y, m, k) = gs(x, y, m, k)) ∧ (c(x, y, m, k) = gc(x, y, m, k)))}.
Cell_I = {p0(x, y, m, k) = (x ∧ y),
p1(x, y, m, k) = (p0(x, y, m, k) ⊕ m),
p2(x, y, m, k) = (p0(x, y, m, k) ∧ m),
p3(x, y, m, k) = (p1(x, y, m, k) ∧ k),
s(x, y, m, k) = (p1(x, y, m, k) ⊕ k),
c(x, y, m, k) = (p3(x, y, m, k) ∨ p2(x, y, m, k))}.
Cell_S = {g0(x, y, m, k) = (x ∧ y),
gs(x, y, m, k) = (g0(x, y, m, k) ⊕ m ⊕ k),
gc(x, y, m, k) = ((g0(x, y, m, k) ∧ m) ∨ (g0(x, y, m, k) ∧ k) ∨ (m ∧ k))}.



3.5. Definition. • Let P be a regular program. We denote with stdsol(P) the standard solution to P. By 3.3 stdsol(P) is well defined. Let p be a predicate symbol in P. We write bdd(P, p) for bdd(stdsol(P)(p)).

• A *query* is a pair (P, g), where P is a regular program and g is a predicate symbol in P. Answer_Query is a function from queries to Boole s.t. Answer_Query(P, g) = 1 iff stdsol(P) ⊨ $g(x_1, \ldots x_n)$. Thus Answer_Query(P, g) = Answer_MC(stdsol(P), $g(x_1, \ldots x_n)$). We call Answer_Query(P, g) the answer to query (P, g) and the problem of computing Answer_Query the query (evaluation) problem. Note that in this situation P is often seen as a deductive database (see [15] sec. 21).

3.6. Example. A verification task can be cast as a query problem. E.g. the circuit in Fig.3.1 is represented with program Cell_I (in Fig.3.1). To check if Cell_I satisfies the specification in Cell_S (Fig.3.1) amounts to answer the query (P, g), where P is defined in Fig.3.1.

Fig.3.2. M(s0)(x) = (**if** x **then** 1 **else** 0),
M(W)(x, z) = (**if** x **then** 1 **else** z),
M(r)(x, z) = (**if** x **then** z **else** 1),
φ ≡ (s0(x) ∧ μh[λ x, z[r(x, z) ∨ ∃ v [h(x, v) ∧ r(v, z)]]](x, z)) → W(x, z),
P_M = {s0(x) = x, W(x, z) = (x ∨ (¬x ∧ z)), r(x, z) = ((x ∧ z) ∨ ¬x)},
P_φ = {g(x, z) = ((s0(x) ∧ h(x, z)) → W(x, z)), h(x, z) = (r(x, z) ∨ ∃ v (h(x, v) ∧ r(v, z)))},
$P = P_M \cup P_\varphi$.

Fig.3.3. M(s)(x) = (**if** x **then** 1 **else** 0),
φ ≡ s(z) ∨ μp[λ x[p(x) ∨ s(z) ∨ ¬ μq[λ y[q(y) ∨ ¬ p(y)]](x)]](x),
P_M = {s(x) = x},
P_φ = {g(x, z) = (s(z) ∨ p(x, z)), p(x, z) = (p(x, z) ∨ s(z) ∨ ¬ q(x, z)), q(x, z) = (q(x, z) ∨ ¬ p(x, z))}, $P = P_M \cup P_\varphi$.

Theorem 3.7 shows that queries are as expressive as μ-calculus MC on a boolean domain.

3.7. Theorem. 0. Let (P, g) be a query. We can find a μ-calculus interpretation M and a μ-calculus formula φ s.t. M ⊨ φ iff stdsol(P) ⊨ $g(x_1, \ldots x_n)$.

1. Let M be a μ-calculus interpretation and φ be a μ-calculus formula. We can find a query (P, g) s.t. M ⊨ φ iff stdsol(P) ⊨ $g(x_1, \ldots x_n)$. Moreover $P = P_M \cup P_\varphi$, where: P_M is a regular program s.t. Alph(P_M) is the set of predicate symbols free in φ and size(P_φ) = O(size(φ))

(size(φ) is the number of symbols in φ).

Proof. (*Sketch*) 0. Trivial. Take $\varphi \equiv g(x_1, \ldots x_n)$ and M = stdsol(P).

1. Defining M and φ in P (see example 3.8). See appendix B for more details.

3.8. Example. Let M, φ, P, P_M, P_φ be as in Fig.3.2 or Fig.3.3. Then $M \models \varphi$ iff stdsol(P) $\models g(x, z)$.

Since we are on a Boolean Domain stdsol(P) and Answer_Query are computable. However to get an automatic verifier we need to show that Answer_query is efficiently computable. Theorem 3.9 shows that query evaluation is at least as efficient as μ-calculus MC via BDDs. In particular we have BDD based algorithms for query evaluation.

3.9. Theorem. Let (P, g) be a query and let Alph(P) = $\{p_1, \ldots p_k\}$. There are BDD based algorithms bdd_compile, bdd_eval s.t.:

- bdd_compile(P) = (bdd(P, p_1), ... bdd(P, p_k)).
- Answer_Query(P, g) = bdd_eval(bdd(P, p_1), ... bdd(P, p_k), g) = bdd_eval(bdd_compile(P), g).

Proof. (*Sketch*) Function bdd_compile is obtained implementing with BDDs the algorithm in the proof of 3.3. E.g. if Card(P) = 1 then bdd_compile is similar to FIXEDPOINT in [2] sec. 4. Since $g \in$ Alph(P) bdd_eval simply tests if bdd(P, g) represents the boolean function identically equal to 1. This is done in constant time since BDDs are a canonical representation for boolean functions.

3.10. Remark. 0. From theorem 3.9 follows that all MC algorithms can be *imported* in a query evaluation framework. E.g. to improve fixpoint computations we can use [2] sec. 5, [11], [16], [13] in (the proof of 3.3 and thus in) bdd_compile in 3.9.

1. Regular programs and queries define a BFOL based functional programming language that we call BFP (*Boolean Functional Programming*). Theorem 3.3 and Answer_Query (3.5) define a semantics for BFP. Theorem 3.9 defines a BDD based compiler for BFP. Note that a regular program is (the completion of) a logic program (see [15] sec. 17). Thus syntactically BFP is a *Logic Programming Language*. However computationally BFP is a *Functional Programming Language*. In fact query evaluation is carried out with a BDD based computation (bdd_eval). Thus for BFP BDDs play the same role as λ-calculus in *traditional* functional programming.

2. From theorem 3.7 follows that a μ-calculus MC problem (M, φ) can be cast as a query problem (P, g). Thus, using queries, systems and system specifications are represented (in P) using the same language (namely BFOL). We will exploit this feature by means of *program transformations*.

4. Program Transformations

So far we have just shown that query evaluation is as good as MC. However the interest of query evaluation for automatic verification relies on the possibility of using *program transformations* to avoid state explosion. This makes query evaluation more efficient than MC. A program transformation is a map taking a query (P, g) and returning an (hopefully) *easier* query (P', g') s.t. Answer_Query(P, g) = Answer_Query(P', g'). We present (4.2, 4.3) an easy and effective program transformation.

4.0. Example. Our program transformation attempts to decompose a program (system) into independent subprograms (subsystems). An informal example will help. Let P0 be as in Fig.3.1 and let (P1, f1), (P2, f2) be as in Fig.4.0. We have: Answer_Query(P1, f1) = Answer_Query(P2, f2) since stdsol(P0) $\models$ c(x, y, m, k) = gc(x, y, m, k). However computing Answer_Query(P2, f2) is easier than computing Answer_Query(P1, f1). In fact BDDs for stdsol(P2)(f), stdsol(P2)(g) are smaller than BDDs for stdsol(P1)(f), stdsol(P1)(g) since P2 avoids composition with stdsol(P2)(c) (= stdsol(P2)(gc)). Moreover if Answer_Query(P2, f1) = 1 then Answer_Query(P2, f2) = 1. Since Answer_Query(P2, f1) = 1 holds we do not even need to compute Answer_Query(P2, f2). We simply compute Answer_Query(P2, f1), a stronger property which is, however, computationally easier to verify.

Fig.4.0. P0 is as in Fig.3.1.

Z ≡ x, y0, m0, k, y1, m1.

P1 = P11 ∪ P0 ∪ {f1(Z) = (f(Z) = g(Z))},

P2 = P22 ∪ P0 ∪ {f1(Z, u) = (f(Z, u) = g(Z, u)),
f2(Z, u) = ((u = c(x, y0, m0, k)) → f1(Z, u))},

P3 = P33 ∪ P0 ∪ {f1(Z, u) = (f(Z, u) = g(Z)),
f3(Z, u) = ((u = c(x, y0, m0, k)) → f1(Z, u))},

P11 = {f(Z) = ∃ z (s(x, y1, m1, z)
∧ (z = c(x, y0, m0, k))), g(Z) = ∃ z (gs(x, y1, m1, z)
∧ (z = gc(x, y0, m0, k)))},

P22 = {f(Z, u) = ∃ z (s(x, y1, m1, z) ∧ (z = u)),
g(Z, u) = ∃ z (gs(x, y1, m1, z) ∧ (z = u))},

P33 = {f(Z, u) = ∃ z (s(x, y1, m1, z) ∧ (z = u)),
g(Z) = ∃ z (gs(x, y1, m1, z)
∧ (z = gc(x, y0, m0, k)))}.

To get algorithms from the informal idea in example 4.0 in 4.1, 4.2 we give a few technical definitions.

4.1. Definition. Let P be a program.

- A predicate symbol p in P is said to be *almost GOL* in P iff for each statement $q(x_1, \ldots x_n) = F$ in P, if (p occurs in an atom B occurring in F) then ($B \equiv p(x_1, \ldots x_h)$ and $h \leq n$).
- A predicate symbol p in P is said to be *GOL* (Gate Output Like) in P iff p is almost GOL in P and for each predicate symbol q in P we have: if (q calls p in P) then q is almost GOL in P.

Predicate symbols defining gate outputs are GOL. E.g. the following predicate symbols are GOL: s, g in P in Fig.3.1; s0, W, g in P in Fig.3.2; f, g, c, gc, f1 in P1 in Fig.4.0. Predicate symbols defining operators (e.g. IC's, transition relations, transitive closures) are not GOL. E.g. the following predicate symbols are not GOL: s (IC) in P1 in Fig.4.0; r (transition relation), h (transitive closure) in P in Fig.3.2.

Fig.4.1. Let $S = \{p_1, \ldots p_k\}$. Let $u_1, \ldots u_k$ be fresh variables. Choose arbitrarily functions $\varphi, \theta : \{1, \ldots k\} \rightarrow \{1, \ldots k\}$ s.t.: (for all $i, j \in \{1, \ldots k\}$ [$\varphi(i) = \varphi(j)$ iff $stdsol(P) \models p_i(x_1, \ldots x_{b(i)}) = p_j(x_1, \ldots x_{b(j)})$]) and (for all $i \in \{\varphi(j) \mid j \in \{1, \ldots k\}\}$ [$\varphi(\theta(i)) = i$]).

Note that we can use $bdd(P, p_1), \ldots bdd(P, p_k)$ to test if $stdsol(P) \models p_i(x_1, \ldots x_{b(i)}) = p_j(x_1, \ldots x_{b(j)})$ holds.

- *refreshed*(P, S, q) = **if** ((q $\notin$ S) and (there is no p $\in$ S s.t. p calls q in P) and (q calls S in P)) **then** 1 **else** 0. Note that if *refreshed*(P, S, q) = 1 then q is GOL in P.
- *refresh*(P, S, $\{u_1, \ldots u_k\}$) = P', where P' is obtained from P as follows:

Step 0. For all statements $q(x_1, \ldots x_h) = F$ in P s. t. refreshed(P, S, q) = 1 replace $q(x_1, \ldots x_h) = F$ with $q(x_1, \ldots x_h) = F'$, where F' is obtained from F by replacing, for $i = 1, \ldots k$, $p_i(x_1, \ldots x_{b(i)})$ in F with $u_{\varphi(i)}$.

Step 1. For all atoms $q(x_1, \ldots x_h)$ s.t. refreshed(P, S, q) = 1 replace $q(x_1, \ldots x_h)$ in P with $q(x_1, \ldots x_h, u_1, \ldots u_k)$.

- *reduce_query*(P, g, S) = (*refresh*(P, S, $\{u_1, \ldots u_k\}$) $\cup$ $\{g'(x_1, \ldots x_n, u_1, \ldots u_k) = (G \rightarrow g(x_1, \ldots x_n, u_1, \ldots u_k))\}$, g'), where: (note that *refreshed*(P, S, g) = 1 and that g is GOL in P)

g' is a fresh predicate symbol, $\{\varphi(j) \mid j \in \{1, \ldots k\}\} = \{\rho(1), \ldots \rho(r)\}$, $G \equiv ((u_{\rho(1)} = p_{\theta(\rho(1))}(x_1, \ldots x_{b(\theta(\rho(1)))})) \wedge \ldots (u_{\rho(r)} = p_{\theta(\rho(r))}(x_1, \ldots x_{b(\theta(\rho(r)))})))$.

Note that, strictly speaking, *reduce_query* is not univocally defined. This, however, will be harmless.

A *(P, g)-redex* (4.2.0) is a set S of predicate symbols in P that can be replaced with fresh variables. The new query (P', g') obtained from (P, g) via S is computed from P, g, S by *reduce_query* (4.2.1). A similar idea has been used in λ-calculus to solve systems of equations (e.g. see [5], [6], [18]).

4.2. Definition. 0. Let (P, g) be a query. A nonempty set S of GOL predicate symbols in P is said to be a *(P, g)-redex* iff ((g $\notin$ S) and (g calls S in P) and (there is no p $\in$ S s.t. p calls g in P)).

1. Let (P, g) be a query and S be a (P, g)-redex. Function *reduce_query* is defined as in Fig.4.1. E.g. consider example 4.0 again. Then {c, gc} is a (P1, f1)-redex and (P2, f2) = *reduce_query*(P1, f1, {c, gc}).

Theorem 4.3.0 shows that from a (P, g)-redex S and *reduce_query* we get a program transformation. Theorem 4.3.1 shows that a counter-example (σ) for *reduce_query*(P, g, S) is also a counter-example for (P, g).

4.3. Theorem. Let (P, g) be a query, S be a (P, g)-redex and (P', g') = *reduce_query*(P, g, S).

0. Answer_Query(P, g) = Answer_Query(P', g').
1. For each state σ, if $[stdsol(P'), \sigma](g'(x_1, \ldots x_{n'})) = 0$ then $[stdsol(P), \sigma](g(x_1, \ldots x_n)) = 0$.

Proof. (*Sketch*). Using the following fact. Since all predicates in S are GOL in P we have: (see definition of *reduce_query* in Fig.4.1) for all predicate symbols q in P, if *refreshed*(P, S, q) = 1 then q is GOL in P.

Fig.4.2. P = {p(x, y) = x, q(x, y) = $\neg$p(x, y),
b(x, y) = (q(x, y) $\vee$ q(y, x)),
h(x, y) = ($\neg$x $\vee$ $\neg$y), g(x, y) = (b(x, y) = h(x, y))}.
P' = {p(x, y) = x, q(x, y, u) = $\neg$u,
b(x, y, u) = (q(x, y, u) $\vee$ q(y, x, u)),
h(x, y) = ($\neg$x $\vee$ $\neg$y),
g(x, y, u) = (b(x, y, u) = h(x, y)),
g'(x, y, u) = ((u = p(x, y)) $\rightarrow$ g(x, y, u))}.

4.4. Remark. • (P, g)-redexes avoid function composition by introducing fresh variables. This decomposes a circuit into independent subcircuits. This, in turn, reduces BDD sizes during query evaluation. E.g. from 4.0, 4.2, 4.3 we have: Answer_Query(P2, f2) = Answer_Query(P1, f1).

- Restriction to GOL predicates in 4.2, 4.3 is important. Without it 4.3 fails. E.g. let (P, g), (P', g') be as in Fig.4.2 (note: p, q are not GOL in P). Note that Answer_Query(P, g) = 1. Let σ be a state s.t. $\sigma(x) = \sigma(u)$

= 1, σ(y) = 0. Then [stdsol(P'), σ](g'(x, y, u)) = 0 (but [stdsol(P), σ](g(x, y)) = 1). Thus Answer_Query(P, g) ≠ Answer_Query(P', g').

- (P, g)-redexes only act on predicate symbols that are GOL. Thus non-GOL predicate symbols represent *non-decomposable* subsystems (of the system defined by P). Our program transformation cannot prevent state explosion caused by non-decomposable subsystems. This means that our program transformation is effective on systems obtained connecting together *not-too-large* non-decomposable subsystems. In particular (P, g)-redexes are effective on combinatorial circuits built from *not-too-large* IC's. Note that no MC automatic global optimization technique working on combinatorial circuits is available.
- A verification problem is an MC problem (M, φ) in an MC setting and a query (P, g) in a BFP setting. All algorithms used in an MC setting can be imported in a BFP setting (see 3.10.0). However not all BFP optimization techniques can be easily exported to an MC setting. E.g. 4.2, 4.3 cannot be used in an MC setting since M and φ are defined using different languages. Defining systems and specifications in the same language (BFOL) is the main advantage of using BFP as an automatic verifier.

5. Reduction Strategy

Given a query (P, g) in general not all (P, g)-redexes will improve query evaluation performances. Thus we need an efficient *reduction strategy*, i.e. an algorithm that given a query (P, g) returns a *reasonably good* (P, g)-redex S whithin a *reasonable* time. Note that (if P ≠ NP) we cannot expect to make all queries easy.

In this section we give an efficient reduction strategy (5.4) aimed to avoid function composition in a program. This reduces BDD sizes during query evaluation. Note that this is not the only possible criterion to choose a reduction strategy, but it is an effective one (see sec. 6).

Given a query (P, g) there are $O(2^{size(P)})$ sets of GOL predicates to be considered as *redex candidates*. We cannot consider all of them. Thus we restrict our search to safe redexes (5.1). Example 5.0 motivates restriction to safe redexes.

5.0. Example. Let (P3, f3) = *reduce_query*(P1, f1, {c}) (see Fig.4.0). Computing Answer_Query(P3, f3) is less efficient than computing Answer_Query(P1, f1). In fact to compute Answer_Query(P3, f3) bdd_compile(P3) will create all BDDs created by bdd_compile(P1) plus bdd(P3, f). Such BDD never appears in the computation of Answer_Query(P1, f1) since stdsol(P1)(f) ≠ stdsol(P3)(f). The point is that redex {c} avoids composition with predicate symbol c, but does not avoid composition with function stdsol(P3)(c) since stdsol(P3)(c) = stdsol(P3)(gc).

Fig.5.0. For each predicate p in P arity(P, p) denotes the arity of p in P. To each predicate p in P we associate an integer array *info*(P, p) of size (arity(P, p) + 1) and defined by function *Syntax_Analysis*.

```
integer syntax_var(program P, predicate_symbol p,
integer i) { if (i < arity(P, p)) return (info(P, p)[i + 1]);
             else return (0); }

integer depth(program P, set_of_predicates S) {
    return ( max(info(P, p)[0] | p ∈ S) ); }
/* We write also depth(P, p) for depth(P, {p}) */

void Syntax_Analysis(program P) {
    for each statement p(x0, ... xn - 1) = H in P do {
        info(P, p)[0] = 0;
        for i = 1, ... n do { if (in H there is an occurrence
of xi - 1 which is not in an atom in H) info(P, p)[i] = 1;
                else info(P, p)[i] = 0; }} not_end = 1;
    while (not_end) {not_end = 0;
        for each statement p(x0, ... xn - 1) = H in P do {
/* define depth info */
            temp = max(
                max(info(P, q)[0] + 1 | (q is a GOL
predicate symbol in P) and (q occurs in H) and (q does not
call p in P)),
                max(info(P, q)[0] | (q is a GOL predicate
symbol in P) and (q occurs in H) and (q calls p in P)));
            if (temp > info(P, p)[0]) {not_end = 1;
                info(P, p)[0] = temp;}
/* define syntactic dependency info */
            for i = 1, ... n do {
                for all atoms q(t0, ... tr - 1) in H do {
                    for j = 0, ... r - 1 do {
                        if ((tj ≡ xi - 1) and (info(P, p)[i]
                            < info(P, q)[j])) {not_end =
                            1; info(P, p)[i] = 1; }}}}}}}
```

5.1. Definition. Let (P, g) be a query, S be a (P, g)-redex, (P', g') = *reduce_query*(P, g, S), i be an integer and p be a predicate symbol in P. Let $x_0, x_1, \ldots$ be fresh variables and A ≡ p(x_0, ... x_{n-1}).

0. We say that p depends *semantically* on argument i in P iff not stdsol(P) ⊨ A[x_i := 0] = A[x_i := 1] .

1. We say that p depends *syntactically* on argument i in P iff *syntax_var*(P, p, i) = 1, where *syntax_var* is as in Fig.5.0. Intuitively p depends syntactically on

argument i iff x_i is *involved* in the definition of p in P.

2. S is said to be *safe* iff there exists an integer i satisfying the following conditions:

- For all p ∈ S, p depends semantically on argument i in P';
- g does not depend syntactically on i in P'.

E.g. consider example 4.0 again. Then {c, gc} is a safe (P1, f1)-redex, whereas {c} is not (see 5.0).

The following proposition shows that safe redexes avoid situations like the one described in example 5.0.

5.2. Proposition. Let (P, g) be a query, S be a safe (P, g)-redex and (P', g') = *reduce_query*(P, g, S).

For all p ∈ S, for all GOL predicates q in P, if ((q is g) or (g calls q in P')) then not stdsol(P') ⊨ $p(x_0, \ldots x_{a-1}) = q(x_0, \ldots x_{b-1})$.

Proof. (*Sketch*). Showing that for any regular program P the following facts hold. (0) If p in P depends semantically on argument i in P then p depends syntactically on argument i in P. (1) If p depends semantically on argument i in P and q does not depend syntactically on argument i in P then not stdsol(P) ⊨ $p(x_0, \ldots x_{a-1}) = q(x_0, \ldots x_{b-1})$.

5.3. Remark. We can efficiently decide if a given redex is safe. In fact let (P, g) be a query. We note the following. (0) To efficiently decide if p in P depends semantically on argument i we can use BDDs. It suffices to check if the index corresponding to argument i of p occurs in bdd(P, p). This can be done traversing bdd(P, p). Thus semantic dependency can be decided in time O(size_bdd(bdd(P, p))). (1) To efficiently decide if p in P depends syntactically on argument i we use *Syntax_Analysis* in Fig.5.0 (no BDDs needed).

Unfortunately predicate symbols in a safe redex can easily generate large BDDs. Thus to avoid state explosion we only consider safe redexes built from GOL predicates which definition is not *too complex*. Function *depth* (see Fig.5.0) defines our measure of complexity for predicate symbol definitions. We only consider safe redexes which predicate symbols have depth less than a given integer. Theorem 5.4 shows that for such redexes there is an efficient reduction strategy (*reduce_step*).

5.4. Theorem. Let (P, g) be a query, max_depth be an integer and *depth* be as in Fig.5.0. There is a function *reduce_step* s.t.:

0. *reduce_step*(P, g) = (P', g', S), where: (P', g') is a query and S is a set of GOL predicates in P.

1. If *reduce_step*(P, g) = (P', g', S) and S is not empty then S is a safe (P, g)-redex s.t.: (P', g') = *reduce_query*(P, g, S) and (*depth*(P, S) < max_depth).

2. If *reduce_step*(P, g) = (P', g', S) and S is empty then P' = P and g' ≡ g.

3. *reduce_step* is efficient. In particular it runs in time Polynomial in T_bdd, max_bdd, size(P), where: T_bdd is the time to create all BDDs in **G**, **G** = {bdd(P, p) | p is a GOL predicate symbol in P s.t. *depth*(P, p) < max_depth}, max_bdd = max(size_bdd(L) | L ∈ **G**).

4. *reduce_step* can be used to compute Answer_Query.

Proof. (*Sketch*) From remark 5.3, the algorithms sketched in Fig.5.1 and *reduce_query* in Fig.4.1.

5.5. Remark. Our reduction strategy (*reduce_step* in Fig.5.1) and our program transformation (*reduce* in Fig.5.1) use at the same time BDDs (e.g. to test for semantic dependency and to compute *reduce_query*) and syntactic analysis (e.g. to test for syntactic dependency). The possibility of using *simultaneously* Logic (syntax) and Models (via BDDs) is the main advantage of BFP w.r.t MC and BDDs alone.

Fig.5.1.

```
boolean Answer_Query(query (P, g)) {
    (P, g) = reduce(P, g);
    return (bdd_eval(bdd_compile(P), g)); }

query reduce(query (P, g)) {(P, g, S) = reduce_step(P, g);
    while (S is not empty) (P, g, S) = reduce_step(P, g);
    return (P, g); }

triple reduce_step(query (P, g)) {
    if (g is not GOL in P) return (P, g, empty_set);
    Syntax_Analysis(P);
    Good = {p | p is a GOL predicate symbol in P and
        depth(P, p) < max_depth};
    for all p in Good do Semantic_var[p] = {i | the index
        for argument i of p occurs in bdd(P, p)};
    max_arity = max(arity of p in P | p ∈ Good);
    for i = 0, ... max_arity - 1 do {
        S = {p | (p ∈ Good) and (i ∈ Semantic_var[p])};
        if (S is a (P, g)-redex) {
            (P', g') = reduce_query(P, g, S);
            Syntax_Analysis(P');
            if (syntax_var(P', g, i) == 0)
/* safe redex found */          return (P', g', S); }}
/* no safe redex found */
    return (P, g, empty_set); }
```

---

6. Experimental results

To assess practical usefulness of our program

transformation we need to study its behavior on interesting practical cases. We are implementing (in C) a compiler for BFP. So far we only have a partial implementation. However it is sufficient to carry out some experiments. We verified a commercial multiplier based on Motorola 2 bit multiplier MC14554B ([17]). We used the constructor suggested expansion diagram to obtain multipliers of larger sizes. Fig.6.0 reports times for computing Answer_Query as defined in Fig.5.1. Syntactic analysis has been done separately since integration between it and BDDs is not yet completed. Adding syntactic analysis will only make verification longer (see function *Syntax_Analysis* in Fig.5.0). As well known ([4]) multipliers of our sizes are out of reach for BDDs alone because of exponential explosion. Using BFP (and *reduce* in Fig.5.1) we get BDDs of modest sizes.

Fig.6.0.

Multiplier Size	Run Time (sec)	Larger BDD
4×4	1.5	844
8×8	4	3375
16×16	17	12700
32×32	68	46967
64×64	335	79989
max_depth = 4;	machine = SUN Sparc Station 2.	

7. Conclusions and further developments

We presented (sec. 3) a BFOL based functional programming language (BFP) that allows symbolic execution of BFOL specifications. For such language we defined (3.9) a BDD based compiler. Using BFP finite state systems and their specifications are defined using the same language (BFOL). This allows *simultaneous* use of Logic (syntax) and Models (via BDDs) to carry out automatic verification. This is the strength of BFP w.r.t. Model Checking. We exploit this feature by defining (4.2, 4.3, 5.4) a *program transformation* reducing function composition in a program. This, in turn, reduces BDD sizes during verification. Using BFP we were able to automatically verify a 64 bit commercial multiplier (sec. 6). A task out of reach for BDDs alone.

Syntactically BFP is a *Boolean Logic Programming* language, but computationally it behaves as a second order *Boolean Functional Programming* language. This allows importing in BFP of ideas and algorithms from FOL theorem proving and functional programming as well. As a matter of fact our program transformation is already an example in this direction (4.2). Moreover BFP can be easily *coupled* with a FOL theorem prover. This is because BFP uses BFOL and BFOL can be seen as a sublanguage of FOL. We are currently implementing (in C) a compiler for BFP and studying more powerful program transformations.

Acknowledgments

I am grateful to Edmund Clarke for very useful discussions on Model Checking and BDDs we had when I was a graduate student at CMU. Those discussions shaped my view of this subject.

I am also grateful to Rocco De Nicola and Rosario Pugliese for very helpful comments on a preliminary version of this paper.

Appendix

A. Proof of 3.3

We sketch the proof of 3.3. We give details only for the constructive part of it.

By induction on Card(clusters(P)) we can prove that if I and J are standard solutions to P then I = J.

To give an algorithm to compute a (the) standard solution to P we need a few technical definitions.

Fig.A.0. interpretation omega(map T) {
old = empty_set; new = T(old);
while (old ≠ new) {old = new; new = T(new);}
return (old); }

A.0. Definition. • Let T be a map from interpretations to interpretations. We define interpretation omega(T) as in Fig.A.0. By Tarsky-Knaster fixpoint theorem (e.g. see [15] sec. 5.1) we have that if T is monotonic then omega(T) terminates returning the least fixpoint of T.

• Let G be a module. We define: import(G) = {g | (g is a predicate symbol in G) and (g does not occur in the head of any statement in G)}.

• Let G ∈ clusters(P) and J be an interpretation. We define clusol(J, G) as follows: (J' = J(import(G))

If G = {$p(x_1, \ldots x_n) = F$} then clusol(J, G) = omega(T), where T(I) = {$p(\sigma(x_1), \ldots \sigma(x_n))$ | [J' ∪ I, σ](F) = 1};

If G = {$p_1(x_1, \ldots x_{a(1)}) = F_1, \ldots p_n(x_1, \ldots x_{a(n)}) = F_n$}, n > 1 and $p_1 < \ldots < p_n$ then clusol(J, G) = I1 ∪ clusol(J' ∪ I1, H), where: H = G - {$p_1(x_1, \ldots x_{a(1)}) = F_1$}, T(I) = {$p_1(\sigma(x_1), \ldots \sigma(x_n))$ | [J' ∪ I ∪ clusol(J' ∪ I, H), σ](F1) = 1}, I1 = omega(T).

From the regularity of P, Tarsky-Knaster fixpoint theorem and the fact that we are on a Boolean Domain we can prove that clusol is well defined.

- We define the interpretation sol(P) as follows:

If clusters(P) = {G} then sol(P) = clusol(∅, G);

If Card(clusters(P)) > 1 then sol(P) = sol(P - G) ∪ clusol(sol(P - G), G), where: G is an arbitrarily chosen module in clusters(P) s.t. there is no p in (P - G) s.t. p calls export(G) in P. Note that Card(clusters(P - G)) < Card(clusters(P)), thus sol(P) is well defined.

We can prove that sol(P) in A.0 is indeed a standard solution to P. This concludes the proof of 3.3.

B. Proof of 3.7.1

We sketch the proof of 3.7.1. We give details only for the constructive part of it. We follow μ-calculus definition in [2]. Note that we call individual variables and relational variables in [2], respectively, variables and predicate symbols. Let free_pred(φ) be the set of predicate symbols free in φ. W.l.o.g. we can assume that M = M(free_pred(φ)) (see 2.1). Thus M defines a finite set of boolean functions. Since boolean functions can be defined using regular programs we can easily find a regular program P_M s.t. stdsol(P_M) = M. To define P_φ we need a few technical definitions.

B.0. Definition. • Let FH be a map from natural numbers to predicate symbols not occurring in φ or P_M s.t.: for all i, k ∈ ℕ, if i < k then FH(i) < FH(k).

- Let mu_bfp be the function defined in Fig.B.0.

Fig.B.0. We call bfp_triple a triple (i, P, φ) where: i ∈ ℕ, P is a finite set of program statements and φ is a μ-calculus formula.

```
bfp_triple mu_bfp(bfp_triple (i, P, φ)) {
    if (φ ≡ p(x1, ... xn) and p is a predicate symbol)
return (i, P, p(x1, ... xn));
    if (φ ≡ λ x1, ... xn[F](y1, ... yn)) {let F' be obtained
from F by simultaneously replacing x1, ... xn in F with
y1, ... yn; return (mu_bfp(i, P, F')); }
    if (φ ≡ μp[R](x1, ... xn)) {let FV(φ) = {z1, ... zk,
x1, ... xn}; p' = FH(i); let R' be obtained from R by
replacing all occurrences of p(t1, ... tn) in R with p'(t1,
... tn, z1, ... zk); (i', P', F) = mu_bfp(i + 1, P, R'(x1, ...
xn)); return (i', P' ∪ {p'(x1, ... xn, z1, ... zk) = F}, p'(x1,
... xn, z1, ... zk)); }
    if (φ ≡ ¬ F) { (i', P', F') = mu_bfp(i, P, F); return (i',
P', ¬ F'); }
    if (φ ≡ F ∨ G) { (i1, P1, F1) = mu_bfp(i, P, F); (i2,
P2, G1) = mu_bfp(i1, P1, G); return (i2, P2, F1 ∨ G1); }
    if (φ ≡ ∃ z [F]) { (i', P', F') = mu_bfp(i, P, F); return
(i', P', ∃ z [F']); } }
```

Let (i, P1, F) = mu_bfp(0, ∅, φ). Let FV(F) = {x_1, ... x_n} and g be a fresh predicate symbol. We define: P_φ = P1 ∪ {g(x_1, ... x_n) = F}, P = P_M ∪ P_φ. Finally we can prove 3.7.1 by induction on size(φ).

References

[1] K.R. Apt, *Logic Programming*, Handbook of Theoretical Computer Science, Elsvier 1990

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, *Symbolic model checking: 10^{20} states and beyond*, Information and Computation 98, (1992)

[3] R. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transaction on Computers, Vol. C-35, N.8, Aug. 1986

[4] R. Bryant, On *On the complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication*, IEEE Transaction on Computers, Vol. 40, N.2, Feb. 1991

[5] C. Böhm, E. Tronci, *X-separability and Left-Invertibility in λ-calculus*, LICS 1987, IEEE Computer Society, 1987

[6] C. Böhm, E. Tronci, *About systems of equations, X-separability and Left-Invertibility in λ-calculus*, Information and Computation, 90, (1991)

[7] R. L. Constable et altri, *Implementing Mathematics with the Nuprl Proof Develpment System*, Prentice-Hall, 1986

[8] G. Dowek, A. Felty, G. Huet, C. Paulin-Mohring, B. Werner, *The Coq Proof Assistant user's guide 5.6*, INRIA-Rocquencourt, 134, 1991

[9] E. A. Emerson, *Temporal and Modal Logic*, Handbook of Theoretical Computer Science, Elsvier 1990

[10] M. J. C. Gordon, *HOL: a proof generating System for Higher Order Logic*, in VLSI Specification, Verification and Synthesis, Kluwer SECS 35, Kluwer, 1988

[11] A. J. Hu, D. Dill, *Reducing BDD Size by Exploiting Functional Dependencies*, DAC 93, IEEE Comp. Soc.

[12] A. J. Hu, D. Dill, A. J. Drexler, C. H. Yang, *Higher Level Specification and Verification using BDDs*, CAV 92, to be reprinted in LNCS

[13] D. E. Long, A. Browne, E. M. Clarke, S. Jha, W. R. Marrero, *An Improved Algorithm for the Evaluation of Fixed Point Expressions*, CAV 94, LNCS 818, Springer-Verlag

[14] L. Lamport, *The Temporal Logic of Actions*, Research Report 79, Digital Equipment Corporation, Systems Research Center, Dec. 1991

[15] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag 1987

[16] Y. Matsunaga, P. C. McGeer, R. K. Brayton, *On computing the Transitive Closure of a State Transition Relation*, DAC 93, IEEE Comp. Society.

[17] Motorola, *CMOS Logic Databook*, 1986

[18] E. Tronci, *Equational Programming in λ-calculus*, LICS 1991, IEEE Computer Society, 1991