Efficient Simplification of Bisimulation Formulas

Uffe H. Engberg

BRICS *Department of Computer Science, University of Aarhus, Denmark

Kim S. Larsen †

Department of Mathematics and Computer Science, Odense University, Denmark

Abstract

The problem of checking or optimally simplifying bisimulation formulas is likely to be computationally very hard. We take a different view at the problem: we set out to define a very fast algorithm, and then see what we can obtain. Sometimes our algorithm can simplify a formula perfectly, sometimes it cannot. However, the algorithm is extremely fast and can, therefore, be added to formula-based bisimulation model checkers at practically no cost. When the formula can be simplified by our algorithm, this can have a dramatic positive effect on the better, but also more time consuming, theorem provers which will finish the job.

1 Introduction

The need for validity checking or optimal simplification of first order bisimulation formulas has arisen from recent work on symbolic bisimulation checking of *value-passing calculi* [4, 9, 15]. The NP-completeness of checking satisfiability of propositional formulas [3] implies that validity checking of that class of formulas is co-NP complete. Additionally, checking of quantified formulas is P-space hard [7], so there is not much hope for a fast algorithm for deciding exactly when a bisimulation formula is valid.

Instead, we set out to solve the problem of what you can get for free, i.e., to what extent is it possible to decide validity simply while reading the formula? As it turns out, there is almost nothing that can be done in linear time. The most simple tasks of storing and retrieving information about variables will cost $O(n \log n)$. So, we allowed ourselves this extra log-factor and changed the question to what you can get *almost* for free. As we shall demonstrate in this paper, the algorithm we have designed is very fast. Not alone does it run in $O(n \log n)$; the constant is also very small. On average, we read through the formulas at a rate of about 75 Kbytes per second. Of course, this is only interesting if the algorithm outputs useful answers reasonably frequently, i.e., in the absence of an obvious notion of optimal simplification (to a minimal equivalent formula), if the algorithm can reasonably frequently guarantee that the formula is valid, rule out that the formula could be valid, or maybe simplify a huge formula to a much smaller equivalent one. It is not easy

* Basic Research in Computer Science, a Centre of the Danish National Research Foundation.

† The initial part of this work was done while this author was at Aarhus University.

to measure how often the algorithm produces a useful answer, but through examples, we show that there are families of process expressions which give rise to formulas, where our algorithm is successful.

The algorithm make a single pass over the formula making no assumptions about the variable names. Notice that this also implies that if formulas are passed on to our algorithm from another program, they do not have to be saved at any point, but can be passed on to our program via pipelining.

As already mentioned, in addition to validity checking, we can also simplify formulas. This greatly increases the usefulness of this work. Even if our algorithm fails to prove the validity of a formula, it will quite often simplify the formula so drastically that the validity of the formula (or the opposite) can easily be asserted by the user. Another possibility is to check the simplified formula for validity using other tools. Tools that would succeed more often, but the complexity of which would make it impossible to work on the original formula. The main advantage of simplifying formulas is that the algorithm can be built into the program generating the bisimulation formulas and simplify the formula on the fly. Also, intermediate simplifications can be used to prune the generation of other subformulas. This can be of vital importance since the formulas can become exponentially large.

Our algorithms work just as well on formulas with free variables as on closed formulas. This means that by simplifying a formula with free variables, we actually characterize the conditions, in terms of the free variables, under which two processes are bisimilar.

2 Bisimulation Formulas

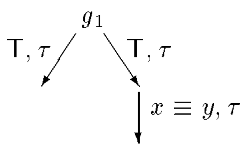
In this section, we introduce bisimulation formulas and some general notions from formal logic. Due to lack of space we shall only briefly sketch how bisimulation formulas are obtained.

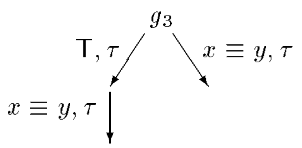
In [4, 9], bisimulation formulas are used in a three stage process of verifying bisimilarity of value-passing programs ([15] obtain similar formulas, albeit with a different approach).

In the first stage, symbolic transition graphs (a generalization of the standard notion of labeled transition graphs) are generated from terms of some value-passing language, say the full CCS calculus [14]. The two graphs are *symbolic bisimilar* [4, 9, 10] iff the two terms are bisimilar in traditional sense.

Below, two processes are shown together with their associated graphs. Following [14], $\mathbf{0}$ is the process having no actions, whereas the prefixed process $\tau.p$ can make an internal action and then act as p. Similarly, there are input (output) prefixes $\alpha?x.$ ($\alpha!x.$) for receiving (sending) values on channel α. The expression **if** E **then** p can do the actions of p, provided the condition E evaluates to true, T. The process $p + p'$ acts as either p or p'.

$A_1(x,y) \stackrel{\text{def}}{=} \tau.\mathbf{0} + \tau.\mathbf{if}\ x \equiv y\ \mathbf{then}\ \tau.\mathbf{0}$ $\qquad$ $A_3(x,y) \stackrel{\text{def}}{=} \tau.(\mathbf{if}\ x \equiv y\ \mathbf{then}\ \tau.\mathbf{0}) + \mathbf{if}\ x \equiv y\ \mathbf{then}\ \tau.\mathbf{0}$





To shorten the presentation, all nodes except for root nodes, have been omitted from the graphs. Each edge is labeled with a guarding condition and a (symbolic) action. The initial conditions of the graph g_1 associated with A_1 are T, whereas the last condition is $x \equiv y$, i.e., x and y should have equivalent values.

In the next stage, an algorithm is used for finding a first order boolean expression mgb, called the most general boolean, characterizing the conditions for which two finite symbolic transition graphs are (symbolic) bisimilar.

Intuitively, two processes are bisimilar if, whenever one process can do an action, the other has a matching action such that the resulting two processes again are bisimilar. This is reflected in the bisimulation formula. For example, the fact that g_3 must match an action corresponding to the left edge of g_1, is captured in the mgb_{g_1,g_3} subformula

$$\mathsf{T} \rightarrow \bigvee \begin{matrix} \mathsf{T} \wedge [\mathsf{T} \wedge (x \equiv y \rightarrow \mathsf{F})] \\ x \equiv y \wedge [\mathsf{T} \wedge \mathsf{T}] \end{matrix}$$

If an instantiation of variables satisfies the guarding condition T of g_1, then it must satisfy a guarding condition, T or $x \equiv y$, of g_3 as well as the corresponding mgb, $[\mathsf{T} \wedge (a \equiv y \rightarrow \mathsf{F})]$ or $[\mathsf{T} \wedge \mathsf{T}]$.

When matching output actions, the values sent must be equal. An equality predicate captures this. By input, universal quantification is used to express that for all values received, the processes are bisimilar.

In the final stage, validity of the bisimulation formula is checked. If it is valid, the original programs are bisimilar under all instantiations. Otherwise, the formula expresses the weakest conditions on the instantiations for which they are bisimilar. Later, in section 7, we shall see that mgb_{g_1,g_3} is in fact valid.

Formally, the class of formulas, which we will work with in the rest of this paper, is defined by the syntax

$$E ::= P \mid E \wedge E \mid E \vee E \mid P \rightarrow E \mid \forall x\colon\ E \qquad \text{and} \qquad P ::= \mathsf{T} \mid \mathsf{F} \mid x \equiv y,$$

where x and y range over a set, V, of variables. As usual, formulas are closed if they have no free variables. Notice that bisimulation formulas only have universal quantification. The binary predicate symbol $\equiv$ is assumed be interpreted as an equivalence relation $\equiv_D$ over a nonempty domain D. It is then standard how to define when an *environment*, i.e., a function from V to D, *satisfies* a formula. An environment satisfies a set of formulas Γ, if it satisfies each formula of Γ. Γ *semantically entails* E, written $\Gamma \models E$, if E is satisfied by any environment satisfying Γ. E is *valid*, $\models E$, if $\emptyset \models E$.

The class of bisimulation formulas is a subset of the class of all quantified formulas and checking validity of bisimulation formulas is not P-space hard. An easy reduction shows that checking satisfiability with respect to an environment is NP complete, so presumably checking validity of bisimulation formulas is co-NP complete.

We now state some general properties of entailment relevant for the development of the algorithm. For simplicity, we write Γ as $E_1, \ldots, E_n$ when $\Gamma = \{E_1, \ldots, E_n\}$. Similarly, we write Γ, Δ for $\Gamma \cup \Delta$.

Theorem 1 (Entailment)

a) If $\Gamma \models E$ then $\Gamma, \Delta \models E$ (Ext)

b) If $E \in \Gamma$ then $\Gamma \models E$ (Rep)

c) If $\Gamma \models E$ and $\Gamma, E \models E'$ then $\Gamma \models E'$ (Cut)

d) If $\Gamma \models E$ and $E \models E'$ then $\Gamma \models E'$ (Trans)

e) $\Gamma \models E$ and $\Gamma \models E'$ iff $\Gamma \models E \wedge E'$ (Conj)

f) $\Gamma, E \models E'$ iff $\Gamma \models E \rightarrow E'$ (Imp)

Proof Standard, see e.g. [16]. □

In general, there is not any similar disjunction theorem allowing both introduction and elimination to the right. However, from the entailment theorems and a few tautologies, we get proposition 2. As a consequence of $\equiv_D$ being an equivalence relation we also have proposition 3.

Proposition 2 If $\Gamma \models E$ or $\Gamma \models E'$, then $\Gamma \models E \vee E'$.

Proposition 3 a) $\models x \equiv x$ b) $x \equiv y \models y \equiv x$ c) $x \equiv y, y \equiv z \models x \equiv z$

3 The Abstract Algorithm

We now set out to design an algorithm for checking validity of formulas with an equivalence predicate. We keep it as abstract as possible to allow for a large degree of freedom in the choice of data structures in the actual implementation.

Intuitively, the idea of the algorithm is to collect in a relation R (over variables) information about variables known to be equivalent when checking subformulas. For instance, checking the validity of a formula like $x \equiv y \rightarrow E$ is reduced to checking E under the assumption that x and y are equivalent, i.e. (x, y) is added to R and E then checked. To exploit that $\equiv$ is an equivalence relation, the symmetric and transitive closure, is taken before proceeding to E. However, when checking E of the formula $\forall x\colon E$ the situation is quite the opposite. Since a new scope is entered, all previous collected information in R concerning x, must be removed before E is checked.

Formally, for R denote the symmetric and transitive closure by $R^{\oplus}$, the reflexive closure, $R \cup \{(x,x) \mid x \in V\}$, by R^0 and the removal of x, $\{(y,z) \in R \mid y \neq x, z \neq x\}$, by $R \setminus x$. Notice, $R \setminus x \subseteq R$, and if R is symmetrically and transitively closed, then so is $R \setminus x$.

In order to connect with the logic, we associate with R the set of formulas $R_{\equiv} \stackrel{\text{def}}{=} \{x \equiv y \mid (x,y) \in R\}$. The notions of closures and removal extend to $R_{\equiv}$ in the natural way: $R_{\equiv} \setminus x$ is $(R \setminus x)_{\equiv}$ etc.

The algorithm is conveniently described using Kleene's three-valued logic [11], the three truth-values being **t** for "true", **f** for "false" and **u** for "undefined"/ "unknown". The Kleene truth tables for conjunction, $\wedge_K$, disjunction, $\vee_K$, and implication, $\rightarrow_K$, are:

$\wedge_K$	t	f	u
t	t	f	u
f	f	f	f
u	u	f	u

$\vee_K$	t	f	u
t	t	t	t
f	t	f	u
u	t	u	u

$\to_K$	t	f	u
t	t	f	u
f	t	t	t
u	t	u	u

The abstract algorithm is expressed in terms of a function, $\Vdash$, which given a bisimulation formula E and a symmetrically and transitively closed relation R, returns $\mathbf{t}$ only if $R_\equiv \models E$, and $\mathbf{f}$ only if $R_\equiv \not\models E$. From now on, R is assumed to be symmetrically and transitively closed. Writing $\Vdash$ infix, the definition is:

$$
R \Vdash E \textbf{ is case } E \textbf{ of } \quad
\begin{array}{rl}
\mathsf{T} & : \mathbf{t} \\
\mathsf{F} & : \mathbf{f} \\
x \equiv y & : \textbf{if } x\ R^0\ y \textbf{ then t else u} \\
E' \wedge E'' & : R \Vdash E' \wedge_K R \Vdash E'' \\
E' \vee E'' & : R \Vdash E' \vee_K R \Vdash E'' \\
E' \to E'' & : R \Vdash E' \to_K R' \Vdash E'', \\
 & \quad \text{where } R' = \begin{cases} (R \cup \{(x,y)\})^\oplus, & \text{if } E' \text{ is } x \equiv y \\ R, & \text{if } E' \text{ is } \mathsf{T} \text{ or } \mathsf{F} \end{cases} \\
\forall x\colon\ E' & : R \setminus x \Vdash E'
\end{array}
$$

Notice that $\Vdash$ is well-defined because we take the symmetric and transitive closure of $(R \cup \{(x,y)\})$.

Given a formula E, the initial call to this function will be $\emptyset \Vdash E$, where $\emptyset$ is the empty relation.

Proving correctness is a matter of proving soundness of $\Vdash$ relative to $\models$. First, we need a small result linking $R_\equiv$ to universal quantification.

Lemma 4 If $R_\equiv \setminus x \models E$, then $R_\equiv \models \forall x\colon\ E$.

Proof Assume that $R_\equiv \setminus x \models E$ and let an environment ρ satisfying $R_\equiv$ be given. Because $R_\equiv \setminus x \subseteq R_\equiv$, ρ must satisfy $R_\equiv \setminus x$ as well. Now x does not occur in any formula of $R_\equiv \setminus x$ so all environments differing from ρ only on the value of x, will then also satisfy $R_\equiv \setminus x$. By the assumption each such environment also satisfies E wherefore the original environment ρ satisfies $\forall x\colon\ E$. $\square$

Writing $\Vdash E$ for $\emptyset \Vdash E$, we can now state the correctness of the algorithm.

Theorem 5 (Correctness) a) If $\Vdash E = \mathbf{t}$, then $\models E$. b) If $\Vdash E = \mathbf{f}$, then $\not\models E$.

Proof Part a) of the theorem follows from the stronger statement

$$\text{if } R \Vdash E = \mathbf{t} \text{ then, } R_\equiv \models E$$

which we prove by induction on the structure of E. Assume $R \Vdash E = \mathbf{t}$. We consider the forms of E:

T, F: In general, $\Gamma \models \mathsf{T}$, so also $R_\equiv \models \mathsf{T}$. The case of F is trivial, since $R \Vdash \mathsf{F} \neq \mathbf{t}$.

$x \equiv y$: By definition of $\_^0$, it follows that $R \models x \equiv y = \mathbf{t}$ iff either $x\ R\ y$ or $x = y$. Now, $x\ R\ y$ is equivalent to $x \equiv y \in R_\equiv$. By (Rep), we get $R_\equiv \models x \equiv y$. In the case $x = y$, the situation is really that E is $x \equiv x$. By (Ext) and a) of proposition 3, we directly obtain $R_\equiv \models x \equiv x$.

$E' \wedge E''$: By definition, $R \models E' \wedge E'' = \mathbf{t}$ implies $R \models E' = \mathbf{t}$ and $R \models E'' = \mathbf{t}$. By induction, we obtain that $R_\equiv \models E'$ and $R_\equiv \models E''$. Using (Conj), we obtain that $R_\equiv \models E' \wedge E''$.

$E' \vee E''$: Similar, using proposition 2 instead of (Conj).

$E' \to E''$: By the definition of $\to_K$, we must have $R \models E' = \mathbf{f}$ or $R' \models E'' = \mathbf{t}$. The forms of E':

- T: Then $R' = R$ and because $R \models \mathsf{T} = \mathbf{t}$, it follows that $R \models E'' = \mathbf{t}$. As above we deduce $R_\equiv \models E''$. Using (Ext), we get $R_\equiv, \mathsf{T} \models E''$ and therefore $R_\equiv \models \mathsf{T} \to E''$ follows by (Imp).
- F: In general $\Gamma, \mathsf{F} \models E$, so in particular $R_\equiv, \mathsf{F} \models E''$. By (Impl), $R_\equiv \models \mathsf{F} \to E''$.
- $x \equiv y$: We have $R \models x \equiv y = \mathbf{t}$ or $R \models x \equiv y = \mathbf{u}$. In either case, we must have $R' \models E'' = \mathbf{t}$, where $R' = (R \cup \{(x, y)\})^\oplus$. By induction, we get $R'_\equiv \models E''$, which is the same as $(R_\equiv, x \equiv y)^\oplus \models E''$. Now any $z \equiv w$ in $(R_\equiv, x \equiv y)^\oplus$ can be deduced from $R_\equiv, x \equiv y$ so by repeated use of (Cut), each of these $z \equiv w$ can be removed from the hypothesis and we finally get $R_\equiv, x \equiv y \models E''$. Thus, by (Impl), $R_\equiv \models x \equiv y \to E''$.

$\forall x\colon E'$: By the induction hypothesis we get that $R_\equiv \setminus x \models E'$. The result follows from lemma 4.

Part b) follows similarly from the stronger statement that if $R \models E = \mathbf{f}$, then $R_\equiv \models \neg E$. □

4 The Concrete Algorithm

In this section, we discuss the implementation of the abstract algorithm outlined in section 3. The function $\models$, which is defined there, closely follows the structure of a formula E. The concrete implementation in this section will follow this structure in exactly the same way. So, the primary task is to find a representation of the relation R such that operations on this relation (union, closure, checking for equivalence, etc.) can be performed efficiently.

The primary operations are to make two variables equivalent and to test whether two variables are already equivalent. This is an instance of the so-called disjoint set problem, which is usually solved using rooted trees [6]. To obtain the best possible performance, path compression (McIllroy and Morris) and union by rank [17] (or similar schemes) are normally used to obtain an amortized complexity of $O(A^{-1}(n))$ per find operation [17, 19], where A^{-1} is the inverse of the (unary) Ackermann function [1].

However, when processing formulas like $(x \equiv y \to E) \wedge E'$, we need to first form the union of the equivalence classes of x and y, then process the expression E, and then deunion (undo) the last union before processing E'. Path compressions are impossible to undo without ruining the complexity, so we only use union by rank, and obtain a complexity of $O(\log n)$ per find [18]. In order to undo the unions, each union operation is registered on a stack. In this way, deunions can be done in constant time (unions are still constant time). These three operations, find, union, and deunion, can also be implemented such that the amortized complexity for the find operation becomes $O(\log n/(\log\log n))$. That proposal is from [12]. See [20] for the analysis. However, the size of the overhead is so large that for formulas that we consider (up to approximately 5Mbytes), this method is slower. For further details on disjoint set implementations, see [13]. We call the structure we use a *union-find-deunion* (UFD) *structure*.

For formulas without universal quantification, this would be all we would need. However, formulas like $(\forall x\colon\ E) \wedge E'$ require that the variable x is freed from previous unions while processing E. Afterwards, for the processing of E', all the old information on x must be restored. Having to keep track of several versions of variables means that the variables cannot be used directly in the *UFD* structure. Instead, we do the following: at any point during the processing of a formula, each variable, x, has an associated stack of pointers corresponding to the number of active versions of x. In greater detail, when a quantifier construction $\forall x\colon$ is encountered, a pointer is pushed onto x's stack. The pointer points to a new item in the *UFD* structure not related to anything, which was previously there. In this way, the old environment can be restored by simply popping the stack.

In order to access the stacks associated with variable names as fast as possible, variable names (along with the pointer to the stacks) are organized in a red-black tree [2, 8], which is one of the efficient implementations of dictionaries with a complexity of $O(\log n)$ per operation, where n is the number of elements in the tree.

To summarize, we use a red-black tree that has variable names as keys and stacks of pointers as values. All these pointers point into a common *UFD* structure. In addition, the *UFD* structure has its own stack of undo information. We refer to the structure consisting of all these other data structures as the *combined* structure.

In the following, we list the operations that the three data structures are assumed to be equipped with. The description is brief as all this is quite well known. However, it seems useful to introduce the names of the operations on the different structures.

A stack is a collection of values, which can be removed from the structure only in the reverse order of which they were inserted. Assume that S is a stack and v is a value. The following operations are supported: $\mathit{Push}(S, v)$, $\mathit{Pop}(S)$, $\mathit{Top}(S)$, $\mathit{Empty}(S)$, and $\mathit{InitStack}()$.

A dictionary implements a set of pairs (k, v), where k is a key value from a totally ordered domain and v is any value. We assume that each key value appears at most once in the dictionary. If T is a dictionary, then the following operations are supported: $\mathit{Insert}(T, k, v)$, $\mathit{Delete}(T, k)$, $\mathit{Member}(T, k)$, $\mathit{LookUp}(T, k)$, and $\mathit{InitTree}()$.

A *UFD* structure is a collection of elements some of which may be equivalent with other elements. The following operations are supported: $\mathit{Union}(U, p, q)$, $\mathit{Find}(U, p)$, $\mathit{Deunion}(U)$, and $\mathit{InitUFD}()$. Obviously, the implementation is basically the well-known union-find structure using a stack to save information about the unions.

A Kleene boolean is an implementation of Kleenes three-valued logic. The three Kleene truth-values *TRUE*, *FALSE*, and *UNKNOWN* correspond to **t**, **f**, and **u**, respectively. The operations *Kand*, *Kor*, and *Kimp* implement the operations $\wedge_K$, $\vee_K$, and $\rightarrow_K$ as described in the tables of section 3. Furthermore, *Ktu* turns an ordinary boolean into the Kleene boolean *TRUE* if it is true and otherwise into *UNKNOWN*.

The Algorithm

In this section, we present the concrete algorithm, which implements the abstract algorithm from section 3. Basically, this is all about representing the relation R using advanced data structures. We assume that the formula E has a representation in the form of a syntax tree. There are well-developed standard techniques to define and manipulate syntax trees. For clarity, we leave out these details.

Also, to present the crucial parts of the algorithm as clearly as possible, we treat $E_1 \wedge E_2$ and $E_1 \vee E_2$ independently. In reality, as we want to process the formula using pipelining, we should process E_1 first and not until after that has been done can we decide whether a conjunction or a disjunction is been processed. Another reasonable assumption would be to require that the program generating the formula does this using a prefix notation like $\wedge(E_1, E_2)$ and $\vee(E_1, E_2)$.

For simplicity, we assume that the formulas are closed, i.e., they do not have any free variables. This is no serious simplification since free variables can be treated as if they were bound at the outermost level.

The concrete implementation follows the structure of the formula in the same way as $\models$, except that the call for the left-hand operand of implication is unfolded and incorporated directly into the case analysis.

```
function check(E: formula) → Kleene boolean;
  var
    r: Kleene boolean;
    p,q: pointers; (* into the UFD structure *)
  case E of
    T:             r := TRUE;
    F:             r := FALSE;
    x ≡ y:         r := Ktu(Find(U,Top(LookUp(T,x))) = Find(U,Top(LookUp(T,y))));
    E1 ∧ E2:       r := Kand(check(E1),check(E2));
    E1 ∨ E2:       r := Kor(check(E1), check(E2));
    T → E1:        r := check(E1);
    F → E1:        r := TRUE;
    (x ≡ y) → E1:  p := Find(U,Top(LookUp(T,x))); q := Find(U,Top(LookUp(T,y)));
                   if p ≠ q then Union(U,p,q);
                   r := Kimp(Ktu(p ≠ q),check(E1));
                   if p ≠ q then Deunion(U);
    ∀x: E1:        if ¬Member(T,x) then Insert(T,x,InitStack());
                   new(p); Push(LookUp(T,x),p);
                   r := check(E1);
                   Pop(LookUp(T,x)); free(p);
                   if Empty(LookUp(T,x)) then Delete(T,x);
  end;
  return r;
end;
```

Before use, T is declared as a red-black tree and properly initialized using InitTree().

Similarly, U is declared as a *UFD* structure and initialized by a call to InitUFD().

Correctness

Proposition 6 The combined structure immediately after a call to the function check is exactly as it were immediately before the call to check.

Proof By induction in the number of calls to the function check. The base case is when this number is one, which means that check is not called recursively. Thus, we must be in one of the cases T, F, or $x \equiv y$. The result follows since the combined structure is not altered in any of these cases.

For the induction step, the result follows trivially from the induction hypothesis in the case where E is $E' \wedge E''$, $E' \vee E''$, $\mathsf{T} \rightarrow E'$, or $\mathsf{F} \rightarrow E'$, since the combined structure is not changed.

Assume that E is $(x \equiv y) \rightarrow E'$. By the induction hypothesis, the call check(E') leaves the structure unchanged. The claim follows as Deunion(U) will undo the last union not yet undone. This must be Union(U,p,q), as the combined structure after the call to check(E') is exactly as it were before the call.

Assume that E is $\forall x:\ E'$. By the induction hypothesis, the call check(E') leaves the structure unchanged. Since LookUp(T,x) is a stack, the statement Pop(LookUp(T,x)) will undo the effect of the statement Push(LookUp(T,x),p). Furthermore, if the stack LookUp(T,x) is empty, then this stack must have been inserted into T by this current invocation of check, so the empty stack should be deleted. $\square$

Proposition 7 Let F be a bisimulation formula, and let E be a subexpression of F with x and y bound in the context of E. Immediately before the call check(E), the combined structure is an exact representation of R in the corresponding call $R \models E$, i.e.,

$$x\ R^0\ y \Leftrightarrow \text{Find(U,Top(LookUp(T,x)))} = \text{Find(U,Top(LookUp(T,y)))}$$

Proof By induction in the structure of E. The base case is when $E = F$, in which case both R (and thus also R^0) and the combined structure are empty. For the induction step, we consider all possible forms that E could have.

If E is T, F, $x \equiv y$, $E' \wedge E''$, $E' \vee E''$, $\mathsf{T} \rightarrow E'$, or $\mathsf{F} \rightarrow E'$, then the combined structure remains unchanged and the same R is used in the recursive application of $\models$.

Assume that E is $(x \equiv y) \rightarrow E'$. Then $\models$ is called with the relation formed by adding (x, y) to R and taking the symmetric and transitive closure. In the combined structure, if x and y do not already belong to the same equivalence class, then the equivalence classes of x and y are joined. Notice that given the representation of the combined structure and the way it is used, it is automatically closed reflexively, symmetrically, and transitively.

Assume that E is $\forall x\colon\ E'$. Then $\models$ is called with the relation formed from R by deleting all pairs that include x. In the combined structure, a new pointer into the *UFD* structure is created and placed on the top of x's variable stack, thus effectively hiding any pairs involving x; except that the pair (x, x) will belong to the structure ensuring reflexitivity. $\square$

Lemma 8 The function, check, correctly implements $\models$.

Proof From proposition 6, it follows that a function semantically equivalent to the function check can be written by letting the combined structure be a value-passing parameter to check. As the two algorithms are structurally equivalent modulo unfolding, it is sufficient to consider the use of the combined structure and R. From proposition 7, it follows that the combined structure is an exact representation of R^0. $\square$

Theorem 9 Let E be a bisimulation formula, and let n be the size of E. Then the time-complexity of check(E) is $O(n \log n)$.

Proof The algorithm is recursive in the structure of E, and clearly, there are a constant number of statements per symbol in E. These statements either perform constant-time operations, or they operate on one of the data structures. As these are initially empty, and as they share the property that n operations are carried out in time $O(n \log n)$, the result follows. $\square$

5 Extensions of the Algorithm

In this section, we consider various extensions of the algorithm. For each extension, we sketch the modifications of the abstract algorithm from section 3, and we discuss the correctness issues briefly.

Many more extension than the ones presented here are possible. However, we have decided only to present extensions according to the criteria:

a) the asymptotic complexity should not change.
b) the increase in the actual complexity should be very low (less than a factor of 10).
c) it should still be a one pass algorithm.

It is not hard to deal with constants and through the obvious transformation suggested by the equivalence

$$(E \wedge E') \rightarrow E'' \quad =\!\models \quad E \rightarrow (E' \rightarrow E''),$$

the algorithm can easily cover implication subformulas with conjunctions of predicates to the left. It is straight forward to cope with multiple equivalence relations by letting the function $\models$ work with multiple relations over variables.

The function $\models$ is only able to return $\mathbf{t}$ ($\mathbf{f}$) if the formula is valid (unsatisfiable). However, the algorithm, $\models_c$, obtained from $\models$ by returning the result of $(R \cup \{(x, y)\})^{\oplus} \models_c E'$ in case of formulas of the form $(x \equiv y) \rightarrow E'$, is able to deal with *contingent* formulas as well.

Theorem 10 a) If $\models_c E = \mathbf{t}$ then $\models E$. b) If $\models_c E = \mathbf{f}$ then $\not\models E$.

Proof The proof of a) is almost exactly as the corresponding proof of theorem 5. Part b) is proved by showing that if $R \models_c E = \mathbf{f}$ then E it is not satisfied by environments identifying all variables. $\square$

$\models_c$ is clearly as good as $\models$. It is also strictly better because $\models x \equiv y \rightarrow \mathsf{F} = \mathbf{u}$ and $\models_c x \equiv y \rightarrow \mathsf{F} = \mathbf{f}$.

6 Simplifications

In this section, we discuss changes to the algorithm with the purpose of outputting a simplified formula equivalent to the original formula. The algorithm should contain the validity checking algorithm as a special case, i.e., if the validity checking algorithm deems a formula valid, then this new algorithm should simplify the formula to T. Also, we would like the algorithm to fulfill the criteria of the previous section.

Like the $\models$ function, the new function, $\models_r$, takes as arguments a relation, R, over variables and a first order formula, but now it returns a first order formula instead of a truth-value of three-valued logic. We use the same case analysis, but turn the Kleene truth tables into simplification tables, essentially by replacing $\mathbf{u}$ by the argument formula. Compare with the Kleene truth tables in section 3. However, this is not quite sufficient. In the $\forall x\colon\ E'$ case of $\models$, $\forall x$ is eliminated completely. This cannot be done here when E' does not simplify to T or F, so a simplification table for $\forall$ is also needed.

$\wedge_r$	T	F	E'
T	T	F	E'
F	F	F	F
E	E	F	$E \wedge E'$

$\vee_r$	T	F	E'
T	T	T	T
F	T	F	E'
E	T	E	$E \vee E'$

$\rightarrow_r$	T	F	E'
T	T	F	E'
F	T	T	T
E	T	$E \rightarrow \mathsf{F}$	$E \rightarrow E'$

$\forall_r\ x\colon$	
T	T
F	F
E'	$\forall x\colon\ E'$

We are now ready to define $\models_r$.

$$
\begin{array}{rll}
R \models_r E \textbf{ is case } E \textbf{ of} & x \equiv y & : \textbf{if } x\ R^0\ y \textbf{ then } \mathsf{T} \textbf{ else } x \equiv y \\
& E' \wedge E'' & : R \models_r E' \wedge_r R \models_r E'' \\
& E' \vee E'' & : R \models_r E' \vee_r R \models_r E'' \\
& E' \rightarrow E'' & : R \models_r E' \rightarrow_r \mathbf{Upd}(R, E') \models_r E'' \\
& \forall x\colon\ E' & : \forall_r\ x\colon\ R \setminus x \models_r E' \\
& E & : E
\end{array}
$$

The final case deals with T and F and for convenience, we have introduced an explicit update function:

$$
\mathbf{Upd}(R, E) = \begin{cases} (R \cup \{(x, y)\})^{\oplus}, & \text{if } E \text{ is } x \equiv y \\ R, & \text{otherwise} \end{cases}
$$

The simplified formula is logically equivalent with the original as stated in the following correctness theorem.

Theorem 11 E if and only if $\models_r E$.

Proof This follows from the statement below which is proved by induction. We omit the details.

$$
R_{\equiv} \rightarrow E \text{ iff } R_{\equiv} \rightarrow (R \models_r E) \qquad \square
$$

The next proposition expresses that $\models_r$ is at least as good as $\models$.

Proposition 12 If $\models E = \mathbf{t}$ ($\mathbf{f}$), then $\models_r E = \mathsf{T}$ (F).

The next section contains examples of simplifications using this algorithm.

A straightforward improvement of the simplification algorithm can be obtained from the semantic equivalence

$$E' \wedge E'' \quad =\!\mid\models \quad E' \wedge (E' \rightarrow E''). \tag{1}$$

Exploiting the simplification of E', the conjunction case is changed to:

$$\textbf{let } E'_r = R \models_r E' \textbf{ in } E'_r \wedge_r \textbf{Upd}(R, E'_r) \models_r E''.$$

In this way, the algorithm can simplify $(F \vee x \equiv y) \wedge (x \equiv y \rightarrow F)$, for example, to F.

Along the same lines, the algorithm can be improved further by using:

$$E \vee E' \quad =\!\mid\models \quad (\neg E) \rightarrow E'.$$

Writing $x \not\equiv y$ for the common occurring formula $x \equiv y \rightarrow \mathsf{F}$, we get as a special case:

$$x \not\equiv y \vee E \quad =\!\mid\models \quad x \equiv y \rightarrow E. \tag{2}$$

As the algorithm is formulated now, there is a priori nothing that prevents the algorithm from working with more predicates such as $x \not\equiv y$ and $x \leq y$. In fact, the simplification algorithm is still sound since the new predicates are not simplified and do not give rise to updates of R through $\textbf{Upd}(\_,\_)$. However, we can use R to simplify the new predicates in some cases, e.g., for $x \not\equiv y$, we can add the case

$$\textbf{if } x\ R^0\ y \textbf{ then } \mathsf{F} \textbf{ else } x \not\equiv y.$$

Now, we turn our attention to another type of simplification. The idea is that universal quantifications can be pushed inwards over conjunctions and that quantified predicates in some cases then can be simplified.

We use this observation to maintain a set, X, of variables corresponding to universal quantified variables met solely by simplification of conjunctions, and define a function $\models_{re}$, which, compared to $\models_r$, takes X as an extra argument.

$$\begin{array}{rrl}
R \models^X_{re} E \textbf{ is case } E \textbf{ of} & x \equiv y & : \textbf{if} \quad x\ R^0\ y \qquad\qquad \textbf{then } \mathsf{T} \\
& & \textbf{elseif } x \in X \textbf{ or } y \in X \textbf{ then } \mathsf{F} \\
& & \textbf{else} \qquad\qquad\qquad\qquad\quad x \equiv y \\
& E' \wedge E'' & : R \models^X_{re} E' \wedge_r R \models^X_{re} E'' \\
& E' \vee E'' & : R \models^\emptyset_{re} E' \vee_r R \models^\emptyset_{re} E'' \\
& E' \rightarrow E'' & : R \models^\emptyset_{re} E' \rightarrow_r \textbf{Upd}(R, E') \models^\emptyset_{re} E'' \\
& \forall x\colon\ E' & : \forall_r\ x\colon\ R \setminus x \models^{X \cup \{x\}}_{re} E' \\
& E & : E
\end{array}$$

The soundness of $\models_{re}$ follows from

$$\begin{array}{rcl}
\forall x\colon\ E \wedge E' & =\!\mid\models & (\forall x\colon\ E) \wedge (\forall x\colon\ E') \\
\forall x\colon\ x \equiv y & =\!\mid\models & \mathsf{F}.
\end{array}$$

Actually, the soundness of the latter requires the quotient set of the domain by the equivalence, $D/{\equiv_D}$, to have a size of at least two. However, for empty or singleton quotient sets,

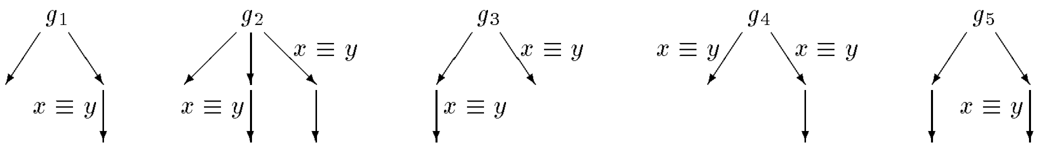
do not seem very useful, so the restriction should not be significant in practice.

Notice that with the exception of the extension concerning contingent formulas, all extensions can all be combined.

7 Examples

In the first half of this section, we focus on qualitative aspects of the simplification algorithm by means of five examples used to illustrate different simplification ideas. In the second half, we deal with some quantitative aspects of the simplification algorithm and the Kleene algorithm through time measures of concrete implementations applied to increasingly larger input.

Consider the following symbolic transition graphs:



All actions are internal, so τ has been omitted from the graphs together with the trivial guarding conditions T. Before proceeding, we invite the reader to try to see which graphs are bisimilar.

Now, applying $\models_r$ to the bisimulation formula mgb_{g_i,g_j} ($\leftrightarrow mgb_{g_j,g_i}$), we get the table of simplified formulas

i \ j	1	2	3	4
2	T			
3	$x \not\equiv y \vee x \equiv y$	$x \not\equiv y \vee x \equiv y$		
4	$x \equiv y \wedge E$	$x \equiv y \wedge E$	E	
5	$x \equiv y \wedge x \not\equiv y$	$(x \equiv y \vee x \equiv y)$ $\wedge\, x \not\equiv y$	$x \equiv y \wedge x \not\equiv y$	$x \equiv y \wedge E$ $\wedge\, x \not\equiv y$

where $E = ((x \equiv y \wedge x \not\equiv y) \vee (x \equiv y \wedge x \equiv y))$, and for sake of readability, $x \equiv y \rightarrow \mathsf{F}$ is written $x \not\equiv y$.

If, in stead, we apply $\models_r$ with the modifications corresponding to (1), many of the formulas are simplified considerably, some even completely as shown in table below to the left.

i \ j	1	2	3	4
2	T			
3	$x \not\equiv y \vee x \equiv y$	$x \not\equiv y \vee x \equiv y$		
4	$x \equiv y$	$x \equiv y$	$x \equiv y$	
5	F	$(x \equiv y \vee x \equiv y)$ $\wedge\, x \not\equiv y$	F	F

i \ j	1	2	3	4
2	T			
3	$x \not\equiv y \vee x \equiv y$	$x \not\equiv y \vee x \equiv y$		
4	F	F	F	
5	F	$(x \equiv y \vee x \equiv y)$ $\wedge\, x \not\equiv y$	F	F

If we are interested in knowing whether the graphs are bisimilar under all instantiations of x and y, we can check validity of the universal closure of the formulas, i.e. simplify the universally closed formulas to T or F if possible. The result of applying $\models_{re}$ (with the modifications mentioned above) yields the table above to the right. We have omitted the quantifiers in the formulas different from T and F.

Adding to $\models_{re}$ an extra case for $\not\equiv$, the formula in entry $(i,j) = (5,2)$ would also simplify to F, and if the modification suggested from (2), i.e., transforming $x \not\equiv y \vee E$ to $x \equiv y \rightarrow E$, is incorporated into the algorithms as well, then the last two entries would also simplify completely, but this time to T.

Turning to the quantitative aspects of the concrete algorithms, we consider processes defined for $i \geq 0$ by

$$p_{i+2} \xrightarrow{\mathsf{T},\alpha?x_0} q_{i+1}, \qquad q_{i+1} \xrightarrow{\mathsf{T},\alpha?x_1} r_1^i, \qquad r_k^{i+1} \xrightarrow{c_k,\alpha?x_{k+1}} r_{k+1}^i \qquad \text{and} \qquad r_k^0 \xrightarrow{c_k,\beta!x_k} \mathbf{0},$$

where c_k is the equality $x_{k-1} = x_k$. Initially two values are unconditionally received on α and then, iteratively, values are received on α provided the two most resently received values are equal. Finally, after i iterations and under the same proviso, the last value is send on β. Similarly, we define primed versions which only differ in that c'_k is the equality $x'_0 = x'_k$. That is, the last value received on α is compared with very first.

In order to give the reader examples of how concrete bisimulation formulas look like, we now describe the most general boolean, $mgb_{r_k^i, r_k'^i}$, characterizing those instantiations (environments) of r_k^i and $r_k'^i$ for which they are late bisimilar.

$$mgb_{r_k^{i+1}, r_k'^{i+1}} = \bigwedge \begin{array}{l} c_k \rightarrow (c'_k \wedge \forall x_{k+1}\colon\ \forall x'_{k+1}\colon\ x_{k+1} = x'_{k+1} \rightarrow mgb_{r_{k+1}^i, r_{k+1}'^i}) \\ c'_k \rightarrow (c_k \wedge \forall x'_{k+1}\colon\ \forall x_{k+1}\colon\ x'_{k+1} = x_{k+1} \rightarrow mgb_{r_{k+1}^i, r_{k+1}'^i}) \end{array}$$

$$mgb_{r_k^0, r_k'^0} = \bigwedge \begin{array}{l} c_k \rightarrow (c'_k \wedge x_k = x'_k \wedge \mathsf{T}) \\ c'_k \rightarrow (c_k \wedge x'_k = x_k \wedge \mathsf{T}) \end{array}$$

The most general booleans for the p's and q's are similar to the first formula above, except that the conditions here are T.

For $2 \leq i \leq 13$, we have measured the average time of five runs of a C implementation of $\models_c$ ($\models_r$) processing $mgb_{p_{i+2}, p'_{i+2}}$ on a SPARC station ELC. To give a few examples, $mgb_{p_{13+2}, p'_{13+2}}$ of size 5.754 Mb was simplified to $\mathbf{t}$ (T) in 77284 (78912) milliseconds. Similarly, $mgb_{p_{13+2}, p'_{13+3}}$, which is 4.477 Mb large, was simplified to $\mathbf{f}$ in 62958 milliseconds. On average, $\models_c$ and $\models_r$ process input at a rate of about 75 Kbytes per second.

References

[1] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Annalen*, 99:118–133, 1928.

[2] R. Bayer. Symmetric Binary B-Trees. *Acta Inform.*, 1:290–306, 1972.

[3] S.A. Cook. The Complexity of Theorem-Proving Procedures. In *ACM STOC*, pages 151–158, 1971.

[4] U.H. Engberg. Simple Symbolic Bisimulations. In preparation.

[5] U.H. Engberg and K.S. Larsen. Efficient Reduction of Bisimulation Formulas. Preprint 47, Dept. of Math. and Computer Science, Odense University, 1993.

[6] B.A. Galler and M.J. Fischer. An improved equivalence algorithm. *Comm. ACM*, 7:301–303, 1964.

[7] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W. H. Freeman, 1979.

[8] L.J.Guibas and R.Sedgewick. A Dichromatic Framework for Balanced Trees. *IEEE FOCS*, 8–21, 1978.

[9] M. Hennessy and H. Lin. Symbolic Bismulations. Tech. Rep. 1/92, University of Sussex, 1992. To appear in *Theoretical Computer Science*, 1995.

[10] M. Hennessy and H. Lin. Proof systems for message-passing process algebras. *CONCUR '93*, pages 202–216, August 1993.

[11] G.J. Klir and T.A. Folger. *Fuzzy Sets, Uncertainty, and Information.* Prentice-Hall, 1988.

[12] H. Mannila and E. Ukkonen. The set union problem with backtracking. *LNCS 226*, 236–243, 1986.

[13] K. Mehlhorn and A. Tsakalidis. Data Structures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 301–341. Elsevier Science Publishers, 1990.

[14] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[15] Z. Schreiber. Verification of Value-Passing Systems. In *First North American Process Algebra Workshop*, pages 9.1–9.20. Tech. Rep. 92-15, Johns Hopkins University, 1992.

[16] D. Scott. Notes on the formalization of logic. Technical report, Sub-faculty of Phil., Oxford, 1981.

[17] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, 22:215–225, 1975.

[18] R.E. Tarjan. *Data Structures and Network Algorithms.* Soc. for Industrial and Applied Math., 1983.

[19] R.E.Tarjan and J.v.Leeuwen. Worst-Case Analysis of Set Union Algorithms. *JACM*, 31:245–281, 1984.

[20] J. Westbrook and R.E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *SIAM J. Comput.*, 18(1):1–11, 1989.