Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests

Damien Pous

Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests

Damien Pous *

Plume team – CNRS, ENS de Lyon, Université de Lyon, INRIA, UMR 5668, France

Damien.Pous@ens-lyon.fr

Abstract

We propose algorithms for checking language equivalence of finite automata over a large alphabet. We use symbolic automata, where the transition function is compactly represented using (multi-terminal) binary decision diagrams (BDD). The key idea consists in computing a bisimulation by exploring reachable pairs symbolically, so as to avoid redundancies. This idea can be combined with already existing optimisations, and we show in particular a nice integration with the disjoint sets forest data-structure from Hopcroft and Karp's standard algorithm.

Then we consider Kleene algebra with tests (KAT), an algebraic theory that can be used for verification in various domains ranging from compiler optimisation to network programming analysis. This theory is decidable by reduction to language equivalence of automata on guarded strings, a particular kind of automata that have exponentially large alphabets. We propose several methods allowing to construct symbolic automata out of KAT expressions, based either on Brzozowski's derivatives or on standard automata constructions.

All in all, this results in efficient algorithms for deciding equivalence of KAT expressions.

Categories and Subject Descriptors F.4.3 [*Mathematical Logic*]: Decision Problems; F.1.1 [*Models of computation*]: Automata; D.2.4 [*Program Verification*]: Model Checking

Keywords Binary decision diagrams (BDD), symbolic automata, Disjoint set forests, union-find, language equivalence, Kleene algebra with tests (KAT), guarded string automata, Brzozowski's derivatives, Antimirov' partial derivatives.

* We acknowledge support from the following ANR projects: 2010-BLAN-0305 PiCoq and 12IS02001 PACE.

To appear in Proc. POPL 15, January 1517 2015, Mumbai, India.
http://dx.doi.org/10.1145/2676726.2677007

1. Introduction

A wide range of algorithms in computer science build on the ability to check language equivalence or inclusion of finite automata. In model-checking for instance, one can build an automaton for a formula and an automaton for a model, and then check that the latter is included in the former. More advanced constructions need to build a sequence of automata by applying a transducer, and to stop whenever two subsequent automata recognise the same language [7]. Another field of application is that of various extensions of Kleene algebra, whose equational theories are reducible to language equivalence of various kinds of automata: regular expressions and finite automata for plain Kleene algebra [26], "closed" automata for Kleene algebra with converse [5, 15], or guarded string automata for Kleene algebra with tests (KAT) [28].

The theory of KAT has been developed by Kozen et al. [12, 27, 28], it has received much attention for its applications in various verification tasks ranging from compiler optimisation [29] to program schematology [3], and very recently for network programming analysis [2, 17]. Like for Kleene algebra, the equational theory of KAT is PSPACE-complete, making it a challenging task to provide algorithms that are computationally practical on as many inputs as possible.

A difficulty with KAT is that the underlying automata work on an input alphabet which is exponentially large in the number of variables of the starting expressions. As such, it renders standard algorithms for language equivalence intractable, even for reasonably small inputs. This difficulty is shared with other fields where various people proposed to work with *symbolic automata* to cope with large, potentially infinite, alphabets [10, 41]. By symbolic automata, we mean finite automata whose transition function is represented using a compact data-structure, typically binary decision diagrams (BDDs) [9, 10], allowing one to explore the automata in a symbolic way.

D'Antoni and Veanes recently proposed a minimisation algorithm for symbolic automata [13], which is much more efficient than the adaptations of the traditional algorithms [22, 31, 32]. However, to our knowledge, the simpler problem of language equivalence for symbolic automata has not been covered yet. We say 'simpler' because language equivalence can be reduced trivially to minimisation—it suffices to minimise the disjoint union of the automata and to check whether the corresponding initial states are equated—but minimisation has complexity $n \ln n$ while Hopcroft and Karp's algorithm for language equivalence [23] is almost linear [40]. (This latter algorithm for checking language equivalence of finite automata can be seen as an instance of Huet's first-order unification algorithm without occur-check [24, Section 5.8]: one tries to unify the two automata recursively, keeping track of the generated equivalence classes of states using an efficient union-find data-structure.)

Our main contributions are the following:

- We propose a simple coinductive algorithm for checking language equivalence of symbolic automata (Section 3). This algorithm is generic enough to support various improvements that have been proposed in the literature for plain automata [1, 6, 14, 42].
- We show how to combine binary decisions diagrams (BDD) and *disjoint set forests*, the efficient data-structure used by Hopcroft and Karp to define their almost linear algorithm [23, 40] for deterministic automata. This results in a new version of their algorithm, for symbolic automata (Section 3.3).
- We study several constructions for building efficiently a symbolic automaton out of a KAT expression (Section 4): we consider symbolic versions of the extensions of Brzozowski's derivatives [11] and Antimirov' partial derivatives [4] to KAT, as well as a generalisation of Ilie and Yu's inductive construction [25]. The latter construction also requires us to generalise the standard procedure consisting of eliminating epsilon transitions.

Notation

We denote sets by capital letters $X, Y, S, T \dots$ and functions by lower case letters $f, g, \dots$ Given sets X and Y, $X \times Y$ is their Cartesian product, $X \uplus Y$ is their disjoint union and X^Y is the set of functions $f\colon Y \to X$. The collection of subsets of X is denoted by $\mathcal{P}(X)$. For a set of letters A, $A^\star$ denotes the set of all finite words over A; ϵ the empty word; and uv the concatenation of words $u, v \in A^\star$. We use 2 for the set $\{0, 1\}$.

2. Preliminary material

We first recall some standard definitions about finite automata and binary decision diagrams.

For finite automata, the only slight difference with the setting described in [6] is that we work with Moore machines [31] rather than automata: the accepting status of a state is not necessarily a Boolean, but a value in a fixed yet arbitrary set. Since this generalisation is harmless, we stick to the standard automata terminology.

2.1 Finite automata

A deterministic finite automaton (DFA) over the input alphabet A and with outputs in B is a triple $\langle S, t, o\rangle$, where S is a finite set of states, $o\colon S \to B$ is the output function, and $t\colon S \to S^A$ is the (total) transition function which returns, for each state x and for each input letter $a \in A$, the next state $t_a(x)$. For $a \in A$, we write $x \xrightarrow{a} x'$ for $t_a(x) = x'$. For $w \in A^\star$, we denote by $x \xrightarrow{w} x'$ the least relation such that (1) $x \xrightarrow{\epsilon} x$ and (2) $x \xrightarrow{au} x'$ if $x \xrightarrow{a} x''$ and $x'' \xrightarrow{u} x'$ for some x''.

The *language* accepted by a state $x \in S$ of a DFA is the function $[\![x]\!]\colon A^\star \to B$ defined as follows:

$$[\![x]\!](\epsilon) = o(x) \ , \qquad [\![x]\!](aw) = [\![t_a(x)]\!](w) \ .$$

(When the output set is 2, these functions are indeed characteristic functions of formal languages). Two states $x, y \in S$ are said to be *language equivalent* (written $x \sim y$) when they accept the same language.

2.2 Coinduction

Checking whether two states of two distinct automata recognise the same language reduces to checking whether two states of a single automaton recognise the same language: one can always build the disjoint union of the two automata. We thus fix a single DFA, and we define bisimulations. We make explicit the underlying notion of progression which we need in the sequel.

```
1  type (s,β) dfa = {t: s → A → s; o: s → β}
2
3  let equiv (M: (s,β) dfa) (x y: s) =
4    let r = Set.empty () in
5    let todo = Queue.singleton (x,y) in
6    while ¬Queue.is_empty todo do
7      (* invariant: r ↣ r ∪ todo *)
8      let (x,y) = Queue.pop todo in
9      if Set.mem r (x,y) then continue
10     if M.o x ≠ M.o y then return false
11     iter_A (fun a → Queue.push todo (M.t x a, M.t y a))
12     Set.add r (x,y)
13   done
14   return true
```

Figure 1. Simple algorithm for checking language equivalence.

Definition 1 (Progression, Bisimulation)**.** *Given two relations $R, R' \subseteq S \times S$ on the states of an automaton,* R progresses to *R', denoted $R \rightarrowtail R'$, if whenever $x \ R \ y$ then*

1. *$o(x) = o(y)$ and*
2. *for all $a \in A$, $t_a(x) \ R' \ t_a(y)$.*

A bisimulation *is a relation R such that $R \rightarrowtail R$.*

Bisimulations provide a sound and complete proof technique for checking language equivalence of DFA:

Proposition 1 (Coinduction)**.** *Two states of an automaton are language equivalent iff there exists a bisimulation that relates them.*

Accordingly, we obtain the simple algorithm described in Figure 1, for checking language equivalence of two states of the given automaton.

This algorithm works as follows: the variable `r` contains a relation which is a bisimulation candidate and the variable `todo` contains a queue of pairs that remain to be processed. To process a pair (x, y), one first checks whether it already belongs to the bisimulation candidate: in that case, the pair can be skipped since it was already processed. Otherwise, one checks that the outputs of the two states are the same ($o(x) = o(y)$), and one pushes all derivatives of the pair to the `todo` queue: all pairs $(t_a(x), t_a(y))$ for $a \in A$. (This requires the type A of letters to be iterable, and thus finite, an assumption which is no longer required with the symbolic algorithm to be presented in Section 3.) The pair (x, y) is finally added to the bisimulation candidate, and we proceed with the remainder of the queue.

The main invariant of the loop (line 7: `r` $\rightarrowtail$ `r` $\cup$ `todo`) ensures that when `todo` becomes empty, then `r` contains a bisimulation, and the starting states were indeed bisimilar. Another invariant of the loop is that for any pair (x', y') in `todo`, there exists a word w such that $x \xrightarrow{w} x'$ and $y \xrightarrow{w} y'$. Therefore, if we reach a pair of states whose outputs are distinct—line 10, then the word w associated to that pair witnesses the fact that the two initial states are not equivalent.

Remark 1. *Note that such an algorithm can be modified to check for language inclusion in a straightforward manner: assuming an arbitrary preorder $\leq$ on the output set B, and letting language inclusion mean $x \leq y$ if for all $w \in A^\star$, $[\![x]\!](w) \leq [\![y]\!](w)$, it suffices to replace line 10 in Figure 1 by*

`if ¬(M.o x ≤ M.o y) then return false`.

2.3 Up-to techniques

The previous algorithm can be enhanced by exploiting *up-to techniques* [36, 39]: an up-to technique is a function f on binary relations such that any relation R satisfying $R \rightarrowtail f(R)$ is contained in a bisimulation. Intuitively, such relations, that are not necessarily bisimulations, are constrained enough to contain only language equivalent pairs.

We have recently shown with Bonchi [6] that the standard algorithm by Hopcroft and Karp [23] actually exploits such an up-to technique: on line 9, rather than checking whether the processed pair is already in the candidate relation `r`, Hopcroft and Karp check whether it belongs to the equivalence closure of `r`. Indeed the function e mapping a relation to its equivalence closure is a valid up-to technique, and this optimisation allows the algorithm to stop earlier. Hopcroft and Karp moreover use an efficient data-structure to perform this check in almost constant time [40]: *disjoint sets forests*. We recall this data-structure in Section 3.3.

Other examples of valid up-to techniques include context-closure, as used in antichain based algorithms [1, 14, 42], or congruence closure [6], which combines both context-closure and equivalence closure. These techniques require working with automata whose states carry a semi-lattice structure, as is typically the case for a DFA obtained from a non-deterministic automaton through the powerset construction.

2.4 Binary decision diagrams

Assume an ordered set $(A, <)$ and an arbitrary set B. Binary decision diagrams are directed acyclic graphs that can be used to represent functions of type $2^A \to B$. When $B = 2$ is the two elements set, BDDs thus intuitively represent Boolean formulas with variables in A.

Formally, a *(multi-terminal, ordered) binary decision diagram* (BDD) is a pair (N, c) where N is a finite set of nodes and c is a function of type $N \to B \uplus (A \times N \times N)$ such that if $c(n) = (a, l, r)$ and either $c(l) = (a', \_, \_)$ or $c(r) = (a', \_, \_)$, then $a < a'$.

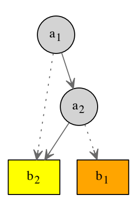
The condition on c ensures that the underlying graph is acyclic, which makes it possible to associate a function $\lceil n \rceil \colon 2^A \to B$ to each node n of a BDD:

$$\lceil n \rceil(\alpha) = \begin{cases} b & \text{if } c(n) = b \in B \\ \lceil l \rceil(\alpha) & \text{if } c(n) = (a, l, r) \text{ and } \alpha(a) = 0 \\ \lceil r \rceil(\alpha) & \text{if } c(n) = (a, l, r) \text{ and } \alpha(a) = 1 \end{cases}$$

Let us now recall the standard graphical representation of BDDs:

- A node n such that $c(n) = b \in B$ is represented by a square box labelled by b.
- A node n such that $c(n) = (a, l, r) \in A \times N \times N$ is a decision node, which we picture by a circle labelled by a, with a dotted arrow towards the *left child* (l) and a plain arrow towards the *right child* (r).

For instance, the following drawing represents a BDD with four nodes; its top-most node denotes the function given on the right-hand side.



$$\alpha \mapsto \begin{cases} b_1 & \text{if } \alpha(a_1) = 1 \text{ and } \alpha(a_2) = 0 \\ b_2 & \text{otherwise} \end{cases}$$

A BDD is *reduced* if c is injective, and $c(n) = (a, l, r)$ entails $l \neq r$. (The above example BDD is reduced.) Any BDD can be transformed into a reduced one. When A is finite, reduced (ordered) BDD nodes are in one-to-one correspondence with functions from 2^A to B [9, 10]. The main interest in this data-structure is that it is often extremely compact.

```
type β node = β descr hash_consed
and β descr = V of β | N of A × β node × β node

val hashcons: β descr → β node
val c: β node → β descr
val memo_rec: ((α'→β'→γ)→α'→β'→γ)→α'→β'→γ
 (* with α' = α hash_consed, β' = β hash_consed *)

let constant v = hashcons (V v)
let node a l r = if l==r then l else hashcons (N(a,l,r))

let apply (f: α → β → γ): α node → β node → γ node =
 memo_rec (fun app x y →
 match c(x), c(y) with
  | V v, V w → constant (f v w)
  | N(a,l,r), V _ → node a (app l y) (app r y)
  | V _, N(a,l,r) → node a (app x l) (app x r)
  | N(a,l,r), N(a',l',r') →
    if a=a' then node a  (app l l') (app r r')
    if a<a' then node a  (app l y ) (app r y )
    if a>a' then node a' (app x l') (app x r'))
```

Figure 2. An implementation of BDDs.

In the sequel, we only work with reduced ordered BDDs, which we simply call BDDs. We denote by $\mathrm{BDD}_A[B]$ the set of nodes of a BDD with values in B, which is large enough to represent all considered functions. We let $\lfloor f \rfloor$ denote the unique BDD node representing a given function $f \colon 2^A \to B$. This notation is useful to give abstract specifications to BDD operations: in the sequel, all usages of this notation actually underpin efficient BDD operations.

Implementation. To better explain parts of the proposed algorithms, we give a simple implementation of BDDs in Figure 2.

The type for BDD nodes is given first: we use Filliâtre's hash-consing library [16] to enforce unique representation of each node, whence the two type declarations and the two conversion functions `hashcons` and `c` between those types. The third utility function `memo_rec` is just a convenient operator for defining recursive memoised functions on pairs of hash-consed values.

The function `constant` creates a constant node, making sure it was not already created. The function `node` creates a new decision node, unless that node is useless and can be replaced by one of its two children. The generic function `apply` is central to BDDs [9, 10]: many operations are just instances of this function. Its specification is the following:

$$\texttt{apply}\ f\ x\ y = \lfloor \alpha \mapsto f(\lceil x \rceil(\alpha))(\lceil y \rceil(\alpha)) \rfloor$$

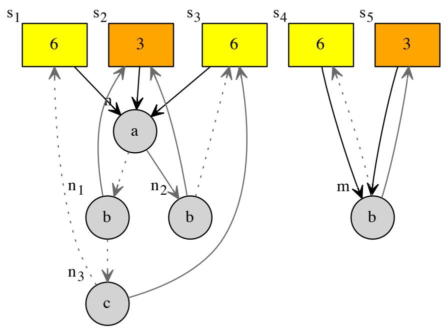
This function is obtained by "zipping" the two BDDs together until a constant is reached. Memoisation is used to exploit sharing and to avoid performing the same computations again and again.

Suppose now that we want to define logical disjunction on Boolean BDD nodes. Its specification is the following:

$$x \vee y = \lfloor \alpha \mapsto \lceil x \rceil(\alpha) \vee \lceil y \rceil(\alpha) \rfloor.$$

We can thus simply use the `apply` function, applied to the Boolean disjunction function:

```
let dsj: bool node → bool node → bool node = apply (||)
```



	s_1, s_2, s_3								s_4, s_5							
a	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
c	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
t	s_1	s_3	s_2	s_2	s_3	s_3	s_2	s_2	s_4	s_4	s_5	s_5	s_4	s_4	s_5	s_5

Figure 3. A symbolic DFA with five states.

Note that this definition could actually be slightly optimised by inlining `apply`'s code, and noticing that the result is already known whenever one of the two arguments is a constant:

```
let dsj: bool node → bool node → bool node =
 memo_rec (fun dsj x y →
 match c(x), c(y) with
   | V true, _ | _, V false → x
   | _, V true | V false, _ → y
   | N(a,l,r), N(a',l',r') →
     if a=a' then node a  (dsj l l') (dsj r r')
     if a<a' then node a  (dsj l y ) (dsj r y )
     if a>a' then node a' (dsj x l') (dsj x r'))
```

We ignore such optimisations in the sequel, for the sake of clarity.

3. Symbolic automata

A standard technique [10, 13, 20, 41] for working with automata over a large input alphabet consists in using BDDs to represent the transition function: a *symbolic DFA* with output set B and input alphabet $A' = 2^A$ for some set A is a triple $\langle S, t, o\rangle$ where S is the set of states, $t\colon S \to \mathrm{BDD}_A[S]$ maps states into nodes of a BDD over A with values in S, and $o\colon S \to B$ is the output function.

Such a symbolic DFA is depicted in Figure 3. It has five states, input alphabet $2^{\{a,b,c\}}$, and natural numbers as output set. We represent the BDD graphically; rather than giving the functions t and o separately, we label the square box corresponding to a state x with its output value $o(x)$ and we link this box to the node $t(x)$ defining the transitions of x using a solid arrow. The explicit transition table is given below the drawing.

The simple algorithm described in Figure 1 is not optimal when working with such symbolic DFAs: at each non-trivial iteration of the main loop, one goes through all letters of $A' = 2^A$ to push all the derivatives of the current pair of states to the queue `todo` (line 11), resulting in a lot of redundancies.

Suppose for instance that we run the algorithm on the DFA of Figure 3, starting from states s_1 and s_4. After the first iteration, `r` contains the pair (s_1, s_4), and the queue `todo` contains eight pairs:

$$(s_1,s_4),(s_3,s_4),(s_2,s_5),(s_2,s_5),(s_3,s_4),(s_3,s_4),(s_2,s_5),(s_2,s_5)$$

```
let iter2 (f: α × β → unit): α node → β node → unit =
  memo_rec (fun iter2 x y →
  match c(x), c(y) with
    | V v, V w → f (v,w)
    | V _, N(_,l,r) → iter2 x l; iter2 x r
    | N(_,l,r), V _ → iter2 l y; iter2 r y
    | N(a,l,r), N(a',l',r') →
      if a=a' then iter2 l l'; iter2 r r'
      if a<a' then iter2 l y ; iter2 r y
      if a>a' then iter2 x l'; iter2 x r')
```

Figure 4. Iterating over the set of pairs reachable from two nodes.

```
1  type (s,β) sdfa = {t: s → s bdd; o: s → β}
2
3  let symb_equiv (M: (s,β) sdfa) (x y: s) =
4    let r = Set.empty() in
5    let todo = Queue.singleton (x,y) in
6    let push_pairs = iter2 (Queue.push todo) in
7    while ¬Queue.is_empty todo do
8      let (x,y) = Queue.pop todo in
9      if Set.mem (x,y) r then continue
10     if M.o x ≠ M.o y then return false
11     push_pairs (M.t x) (M.t y)
12     Set.add r (x,y)
13   done;
14   return true
```

Figure 5. Symbolic algorithm for checking language equivalence.

Assume that elements of this queue are popped from left to right. The first element is removed during the following iteration, since (s_1, s_4) already is in `r`. Then (s_3, s_4) is processed: it is added to `r`, and the above eight pairs are appended again to the queue, which now has fourteen elements. The following pair is processed similarly, resulting in a queue with twenty one $(14 - 1 + 8)$ pairs. Since all pairs of this queue are already in `r`, it is finally emptied through twenty one iterations, and the algorithm returns true.

Note that it would be even worse if the input alphabet was actually declared to be $2^{\{a,b,c,d\}}$: even though the bit d of all letters is irrelevant for the considered DFA, each non-trivial iteration of the algorithm would push even more copies of each pair to the `todo` queue.

What we propose here is to exploit the symbolic representation, so that a given pair is pushed only once. Intuitively, we want to recognise that starting from the pair of nodes (n, m), the letters 010, 011, 110 and 111 are equivalent[1], since they lead to the same pair, (s_2, s_5). Similarly, the letters 001, 100, and 101 are equivalent: they lead to the pair (s_3, s_4).

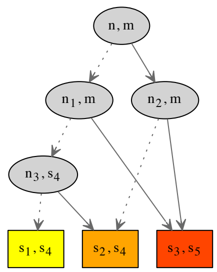
This idea is easy to implement using BDDs: like for the `apply` function (Figure 2), it suffices to zip the two BDDs together, and to push pairs when we reach two leaves. We use for that the procedure `iter2` from Figure 3, which successively applies a given function to all pairs reachable from two nodes. Its code is almost identical to `apply`, except that nothing is constructed (and memoisation is just used to remember those pairs that have already been visited).

We finally modify the simple algorithm from Section 2.1 by using this procedure on line 11; we obtain the code given in Figure 5. We apply `iter2` to its first argument once and for all (line 6), so that we maximise memoisation: a pair of nodes that has been vis-

[1] Letters being elements of $2^{\{a,b,c\}}$ here, we represent them with bit-vectors of length three

ited in the past will never be visited again, since all pairs of states reachable from that pair of nodes are already guaranteed to be processed. (As an invariant, we have that all pairs reachable from a pair of nodes memoised in `push_pairs` appear in $\mathtt{r} \cup \mathtt{todo}$.)

Let us illustrate this algorithm by running it on the DFA from Figure 3, starting from states s_1 and s_4 as previously. During the first iteration, the pair (s_1, s_4) is added to `r`, and `push_pairs` is called on the pair of nodes (n, m). This call virtually results in building the following BDD, where leaves consist of calls to `Queue.push todo`.



The following three pairs are thus pushed to `todo`.

$$(s_1, s_4), (s_3, s_4), (s_2, s_5)$$

The first pair is removed by a trivial iteration: (s_1, s_4) already belongs to `r`. The two other pairs are processed by adding them to `r`, but without pushing any new pair to `todo`: thanks to memoisation, the two expected calls to `push_pairs n m` are skipped.

All in all, each reachable pair is pushed only once to the `todo` queue. More importantly, the derivatives of a given pair are explored symbolically. In particular, the algorithm would execute exactly in the same way, even if the alphabet was actually declared to be much larger (for instance because the considered states were part of a bigger automaton with more letters). In fact, the main loop is executed at most n^2 times, where n is the total number of BDD nodes (both leaves and decision nodes) reachable from the starting states.

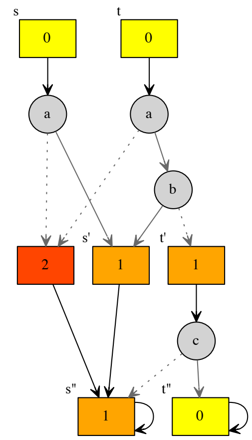
Finally note that in the code from Figure 5, the candidate relation `r` is redundant, as the pairs it contains are also stored implicitly in the memoisation table of `iter2` (except for the initial pair). The corresponding lines (4, 9, and 12) can thus be removed.

3.1 Displaying symbolic counter-examples.

Bisimulation-based algorithms for language equivalence can be instrumented to produce counter-examples in case of failure, i.e., a word which is accepted by one state and not by the other.

An advantage of the previous algorithm is that those counter-examples can be displayed symbolically; thus enhancing readability. This is particularly important in the context of formal assisted proofs (e.g., when working with KAT in Coq [34]), where a plain guarded string is often too big to be useful to the user, while a 'symbolic' guarded string—where only the bits that are relevant for the counter-example are displayed—can be really helpful to understand which hypotheses have to be used to solve the current goal.

Consider for instance the following automaton.



Intuitively, the topmost states s and t are not equivalent because t can take two transitions to reach t'', with output 0, while with two transitions, s can only reach s'', with output 1.

More formally, the word 100 001 over $2^{\{a,b,c\}}$ is a counter-example: we have

$$\begin{aligned} [\![s]\!](100\,001) &= [\![s']\!](001) = o(s'') = 1 \enspace, \\ [\![t]\!](100\,001) &= [\![t']\!](001) = o(t'') = 0 \enspace. \end{aligned}$$

But there are plenty of other counter-examples of length two: it suffices that: a be assigned true and b be assigned false in the first letter, and that c be assigned true in the second letter. The values of the bit c in the first letter, and of the bits a and b in the second letter do not change the above computation. As a consequence, this counter-example is best described as the pseudo-word 10- --1, or alternatively the word $(a \wedge \neg b)\, c$ whose letters are conjunctions of literals indicating the least requirements to get a counter example.

The algorithm from Figure 5 makes it possible to give this information back to the user:

- modify the queue *todo* to store triples (w, x, y) where (x, y) is a pair of states to process, and w is the associated potential counter-example;
- modify the function `iter2` (Figure 3), so that it uses an additional argument to record the encountered node labels, with negative polarity when going through the recursive call for the left child, and positive polarity for the right child;
- modify line 10 of the main algorithm to return the symbolic word associated with the current pair when the output test fails.

3.2 Non-deterministic automata

Standard coinductive algorithms for DFA can be applied to non-deterministic automata (NFA) by using the *powerset construction*. This construction transforms a non-deterministic automaton into a deterministic one; we extend it to symbolic automata in the obvious way.

A *symbolic NFA* is a tuple $\langle S, t, o\rangle$ where S is the set of states, $o\colon S \to B$ is the output function, and $t\colon S \to \mathrm{BDD}_A[\mathcal{P}(S)]$ maps a state and a letter of the alphabet $A' = 2^A$ to a set of possible successor states, using a symbolic representation. The set B of output values must be equipped with a semi-lattice structure $\langle B, \wedge, \bot\rangle$. Assuming such an NFA, one defines a symbolic DFA

$\langle \mathcal{P}(S), t^\sharp, o^\sharp \rangle$ as follows:

$$t^\sharp(\{x_1, \ldots, x_n\}) \triangleq t(x_1) \mathbin{\underline{\cup}} \ldots \mathbin{\underline{\cup}} t(x_n) \enspace ,$$
$$o^\sharp(\{x_1, \ldots, x_n\}) \triangleq o(x_1) \vee \cdots \vee o(x_n) \enspace .$$

(Where $\underline{\cup}$ denotes the pointwise union of two BDDs over sets: $n \mathbin{\underline{\cup}} m = \lfloor \phi \mapsto \lceil n \rceil(\phi) \cup \lceil m \rceil(\phi) \rfloor$.)

This DFA has exponentially many states. However, when applying bisimulation-based algorithms to such automata, one explores them on the fly, and only those subsets that are reachable from the initial states need to be visited. This number of reachable subsets is usually much smaller than the exponential worst-case bound; in fact it is quite often of the same order as the number of states of the starting DFA (see, e.g., the experiments in Section 5).

3.3 Hopcroft and Karp: disjoint sets forests

The previous algorithm can be freely enhanced by using up-to techniques, as described in Section 2.3: it suffices to modify line 9 to skip pairs more or less aggressively, according to the chosen up-to technique. For an up-to technique f, line 9 thus becomes

```
if Set.mem (x,y) (f r) then continue .
```

The up-to-equivalence technique used in Hopcroft and Karp's algorithm can however be integrated in a deeper way, by exploiting the fact that we work with BDDs. This leads to a second algorithm, which we describe in this section.

Let us first recall *disjoint sets forests*, the data structure used by Hopcroft and Karp to represent equivalence classes. This standard data-structure makes it possible to check whether two elements belong to the same class and to merge two equivalence classes, both in almost constant amortised time [40].

The idea consists in storing a partial map from elements to elements and whose underlying graph is acyclic. An element for which the map is not defined is the *representative* of its equivalence class, and the representative of an element pointing in the map to some y is the representative of y. Two elements are equivalent if and only if they lead to the same representative; to merge two equivalence classes, it suffices to add a link from the representative of one class to the representative of the other class. Two optimisations are required to obtain the announced theoretical complexity:

- when following the path leading from an element to its representative, one should compress it in some way, by modifying the map so that the elements in this path become closer to their representative. There are various ways of compressing paths, in the sequel, we use the method called *halving* [40];
- when merging two classes, one should make the smallest one point to the biggest one, to avoid generating too many long paths. Again, there are several possible heuristics, but we elude this point in the sequel.

As explained above, the simplest thing to do would be to replace the bisimulation candidate r from Figure 5 by a disjoint sets forest over the states of the considered automaton.

The new idea consists in relating the BDD nodes of the symbolic automaton rather than just its states (i.e., just the BDD leaves). By doing so, one avoids visiting pairs of nodes that have already been visited up to equivalence.

Concerning the implementation, we first introduce a BDD unification algorithm (Figure 3.3), i.e., a variant of the function `iter2` which uses disjoint sets forest rather than plain memoisation. This function first creates an empty forest (we use Filliâtre's module `Hmap` of maps over hash-consed values to represent the corresponding partial maps). The function `link` adds a link between two representatives; the recursive terminal function `repr` looks for the representative of a node and implements halving. The inner function

```
1  let unify (f: β × β → unit): β node → β node → unit =
2   (* the disjoint sets forest *)
3   let m = Hmap.empty() in
4   let link x y = Hmap.add m x y in
5   (* representative of a node *)
6   let rec repr x =
7    match Hmap.get m x with
8     | None → x
9     | Some y → match Hmap.get m y with
10       | None → y
11       | Some z → link x z; repr z
12  in
13  let rec unify x y =
14   let x = repr x in
15   let y = repr y in
16   if x ≠ y then
17   match c(x), c(y) with
18    | V v, V w → link x y; f (v,w)
19    | V _, N(_,l,r) → link y x; unify x l; unify x r
20    | N(_,l,r), V _ → link x y; unify l y; unify r y
21    | N(a,l,r), N(a',l',r') →
22       if a=a' then link x y; unify l l'; unify r r'
23       if a<a' then link x y; unify l y ; unify r y
24       if a>a' then link y x; unify x l'; unify x r')
25  in unify
```

Figure 6. Unifying two nodes of a BDD, using disjoint set forests.

```
let dsf_equiv (M: (s,β) sdfa) (x y: s) =
 let todo = Queue.singleton (x,y) in
 let push_pairs = unify (Queue.push todo) in
 while ¬Queue.is_empty todo do
  let (x,y) = Queue.pop todo in
  if M.o x ≠ M.o y then return false
  push_pairs (M.t x) (M.t y)
 done;
 return true
```

Figure 7. Symbolic algorithm optimised with disjoint set forests.

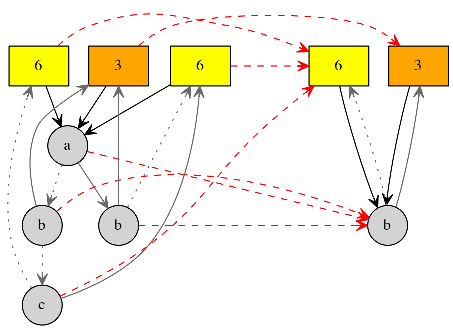
`unify` is defined similarly as `iter2`, except that it first takes the representative of the two given nodes, and that it adds a link from one to the other before recursing.

Those links can be put in any direction on lines 18 and 22, and we should actually use an appropriate heuristic to take this decision, as explained above. In the four other cases, we put a link either from the node to the leaf, or from the node with the smallest label to the node with the biggest label. By proceeding this way, we somehow optimise the BDD, by leaving as few decision nodes as possible.

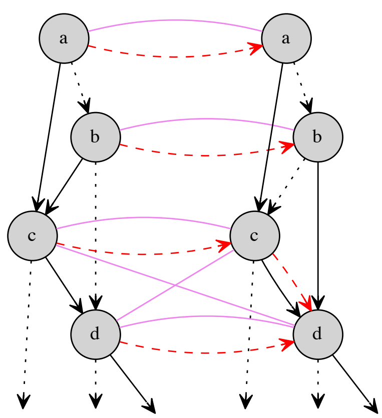
It is important to notice that there is actually no choice left in those four cases: we work implicitly with the optimised BDD obtained by mapping all nodes to their representatives, so that we have to maintain the invariant that this optimised BDD is ordered and acyclic. (Notice that this optimised BDD need not be reduced anymore: the children of given a node might be silently equated, and a node might have several representations since its children might be silently equated with the children of another node with the same label)

We finally obtain the algorithm given in Figure 7. It is similar to the previous one (Figure 5), except that we use the new function `unify` to push pairs into the `todo` queue, and that we no longer store the bisimulation candidate `r`: this relation is subsumed by the restriction of the disjoint set forests to BDD leaves.

If we execute this algorithm on the symbolic DFA from Figure 3, between states s_1 and s_4, we obtain the disjoint set forest depicted below using dashed red arrows. This actually corresponds to the pairs which would be visited by the first symbolic algorithm (Figure 5).



If instead we start from the top-most nodes in the following partly described automaton, we would get the disjoint set forest depicted similarly in red, while the first algorithm would go through all violet lines, one of which is superfluous.



The corresponding optimised BDD consists of the three nodes labelled with a, b, and d on the right-hand side. This BDD is not reduced, as explained above: the node labelled with b should be removed since it points twice to the node labelled with d, and removing this node makes the node labelled with a useless, in turn.

Complexity. Concerning complexity, while the algorithm from Figure 5 is quadratic in the number n of BDD nodes (and leaves) that are reachable from the starting symbolic DFA, the optimised algorithm from Figure 7 performs at most n iterations: two equivalence classes of nodes are merged each time a link is added, and we start with the discrete partition of nodes.

Unfortunately, we cannot immediately deduce that the algorithm is almost linear, as did Tarjan for Hopcroft and Karp's algorithm [40]. The problem is that we cannot always freely choose how to link two representatives (i.e., on lines 19, 20, 23, and 24 in Figure 3.3), so that we cannot guarantee that the amortised complexity of maintaining those equivalence classes is almost constant. We conjecture that such a result holds, however, as the choice we enforce in those cases virtually suppresses binary decision nodes, thus reducing the complexity of subsequent BDD unifications.

Unification with row types. As mentioned in the Introduction, Hopcroft and Karp's algorithm can be seen as an instance of Huet's first-order unification algorithm for recursive terms (i.e., without occur-check). The algorithm presented in Figure 7, and more specifically the BDD unification sub-algorithm (Figure 3.3) is reminiscent of Rémy's extension of this unification algorithm for dealing with *row types*—to obtain an ML-like type inference algorithm in presence of extensible records [33, 37, 38].

More precisely, row types are almost-constant functions from a given set of labels to types, typically represented as association lists with a default value. Unification of such row types is performed pointwise, and is implemented by zipping the two association lists together, as we do here with BDDs (which generalise from almost constant functions to functions with finitely many output values).

It would thus be interesting to understand whether our generalisation of this unification sub-algorithm, from association lists to BDDs, could be useful in the context of unification: either by exploiting the richer structure of functions represented by BDDs, or just for the sake of efficiency, when the set of labels is large (e.g., for type inference on object-oriented programs, where labels correspond to method names).

4. Kleene algebra with tests

Now we consider Kleene algebra with tests (KAT), for which we provide several automata constructions that allow us to use the previous symbolic algorithms.

A *Kleene algebra with tests* is a tuple $\langle X, B, \cdot, +, \cdot^\star, \neg, 1, 0\rangle$ such that:

(i) $\langle X, \cdot, +, \cdot^\star, 1, 0\rangle$ is a Kleene algebra [26], i.e., an idempotent semiring with a unary operation, called "Kleene star", satisfying the following axioms:

$$1 + x \cdot x^\star \leq x^\star$$
$$y \cdot x \leq x \;\Rightarrow\; y^\star \cdot x \leq x$$
$$x \cdot y \leq x \;\Rightarrow\; x \cdot y^\star \leq x$$

(the preorder $(\leq)$ being defined by $x \leq y \triangleq x + y = y$);

(ii) $B \subseteq X$;

(iii) $\langle B, \cdot, +, \neg, 1, 0\rangle$ is a Boolean algebra.

The elements of the set B are called "tests"; we denote them by ϕ, ψ. The elements of X, called "Kleene elements", are denoted by x, y, z. We sometimes omit the operator "$\cdot$" from expressions, writing xy for $x \cdot y$. The following (in)equations illustrate the kind of laws that hold in all Kleene algebra with tests:

$$\phi + \neg\phi = 1 \qquad \phi\cdot(\neg\phi + \psi) = \phi\cdot\psi = \neg(\neg\phi + \neg\psi)$$

$$x^\star x^\star = x^\star \qquad (x{+}y)^\star = x^\star(yx^\star)^\star \qquad (x{+}xxy)^\star \leq (x{+}xy)^\star$$

$$\phi\cdot(\neg\phi\cdot x)^\star = \phi \qquad \phi\cdot(\phi\cdot x\cdot\neg\phi + \neg\phi\cdot y\cdot\phi)^\star\cdot\phi \leq (x\cdot y)^\star$$

The laws from the first line come from the Boolean algebra structure, while the ones from the second line come from the Kleene algebra structure. The two laws from the last line require both Boolean algebra and Kleene algebra reasoning.

Binary relations. Binary relations form a Kleene algebra with tests; this is the main model we are interested in, in practice. The Kleene elements are the binary relations over a given set S, the tests are the predicates over this set, encoded as sub-identity relations, and the star of a relation is its reflexive transitive closure.

This relational model is typically used to interpret imperative programs: such programs are state transformers, i.e., binary relations between states, and the conditions used to define the control-flow of these programs are just predicates on states. Typically, a program "`while` ϕ `do` `p`" is interpreted through the KAT expression $(\phi \cdot p)^\star \cdot \neg\phi$.

KAT expressions. We denote by $\mathcal{R}eg(V)$ the set of *regular expressions* over a set V:

$$x, y ::= v \in V \mid x + y \mid x \cdot y \mid x^\star \ .$$

Assuming a set A of elementary tests, we denote by $B(A)$ the set of *Boolean expressions* over A:

$$\phi, \psi ::= a \in A \mid 1 \mid 0 \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \ .$$

Further assuming a set Σ of letters (or atomic Kleene elements), a *KAT expression* is a regular expression over the disjoint union $\Sigma \uplus B(A)$. We let p, q range over elements of Σ. Note that the constants 0 and 1 from the signature of KAT, and usually found in the syntax of regular expressions, are represented here by injecting the corresponding tests.

Guarded string languages. Guarded string languages are the natural generalisation of string languages for Kleene algebra with tests. We briefly define them.

An *atom* is a valuation from elementary tests to Booleans; it indicates which of these tests are satisfied. We let α, β range over atoms, the set of which is denoted by At: $At = 2^A$. A Boolean formula ϕ is *valid* under an atom α, denoted by $\alpha \vDash \phi$, if ϕ evaluates to true under the valuation α.

A *guarded string* is an alternating sequence of atoms and letters, both starting and ending with an atom:

$$\alpha_1, p_1, \alpha_2, \dots, \alpha_n, p_n, \alpha_{n+1} \ .$$

The concatenation $u * v$ of two guarded strings u, v is a partial operation: it is defined only if the last atom of u is equal to the first atom of v; it consists in concatenating the two sequences and removing one copy of the shared atom in the middle.

To any KAT expression, one associates a *guarded string language*, i.e., a set of guarded strings, as follows.

$$\begin{aligned}
G(\phi) &= \{\alpha \in At \mid \alpha \vDash \phi\} && (\phi \in B(A))\\
G(p) &= \{\alpha p \beta \mid \alpha, \beta \in At\} && (p \in \Sigma)\\
G(x + y) &= G(x) \cup G(y)\\
G(xy) &= \{u * v \mid u \in G(x), v \in G(y)\}\\
G(x^\star) &= \{u_1 * \dots * u_n \mid \exists u_1 \dots u_n, \forall i \leq n, u_i \in G(x)\}
\end{aligned}$$

KAT Completeness. Kozen and Smith proved that the equational theory of Kleene algebra with tests is complete over the relational model [30]: any equation that holds universally in this model can be proved from the axioms of KAT. Moreover, two expressions are provably equal if and only if they denote the same language of guarded strings. By a reduction to automata theory this gives algorithms to decide the equational theory of KAT. Now we study several such algorithms, and we show each time how to exploit symbolic representations to make them efficient.

4.1 Brzozowski's derivatives

Derivatives were introduced by Brzozowski [11] for (plain) regular expressions; they make it possible to define a deterministic automaton where the states of the automaton are the regular expressions themselves.

Derivatives can be extended to KAT expressions in a very natural way [28]. We recall this extension in Figure 8: one first defines a Boolean function ϵ_α, that indicates whether an expression accepts

$$\begin{aligned}
\epsilon_\alpha(x+y) &= \epsilon_\alpha(x) + \epsilon_\alpha(y) & \delta_{\alpha p}(x+y) &= \delta_{\alpha p}(x) + \delta_{\alpha p}(y)\\
\epsilon_\alpha(x \cdot y) &= \epsilon_\alpha(x) \cdot \epsilon_\alpha(y) & \delta_{\alpha p}(x \cdot y) &= \begin{cases} \delta_{\alpha p}(x) \cdot y & \text{if } \epsilon_\alpha(x) = 0\\ \delta_{\alpha p}(x) \cdot y + \delta_{\alpha p}(y) & \text{oth.}\end{cases}\\
\epsilon_\alpha(x^\star) &= 1 & \delta_{\alpha p}(x^\star) &= \delta_{\alpha p}(x) \cdot x^\star\\
\epsilon_\alpha(q) &= 0 & \delta_{\alpha p}(q) &= \begin{cases} 1 & \text{if } p = q\\ 0 & \text{oth.}\end{cases}\\
\epsilon_\alpha(\phi) &= \begin{cases} 1 & \text{if } \alpha \vDash \phi\\ 0 & \text{oth.}\end{cases} & \delta_{\alpha p}(\phi) &= 0
\end{aligned}$$

Figure 8. Explicit derivatives for KAT expressions

$$\begin{aligned}
\epsilon^{s}(x+y) &= \epsilon^{s}(x) \vee \epsilon^{s}(y) & \delta^{s}(x+y) &= \delta^{s}(x) \oplus \delta^{s}(y)\\
\epsilon^{s}(x \cdot y) &= \epsilon^{s}(x) \wedge \epsilon^{s}(y) & \delta^{s}(x \cdot y) &= (\delta^{s}(x) \odot y) \oplus (\epsilon^{s}(x) \otimes \delta^{s}(y))\\
\epsilon^{s}(x^\star) &= 1 & \delta^{s}(x^\star) &= \delta^{s}(x) \odot x^\star\\
\epsilon^{s}(p) &= 0 & \delta^{s}(p) &= \lceil p \mapsto 1, _ \mapsto 0 \rceil\\
\epsilon^{s}(\phi) &= \phi & \delta^{s}(\phi) &= 0
\end{aligned}$$

Figure 9. Symbolic derivatives for KAT expressions

the single atom α; this function is then used to define the derivation function $\delta_{\alpha p}$, that intuitively returns what remains of the given expression after reading the atom α and the letter p. These two functions make it possible to give a coalgebraic characterisation of the characteristic function of G. We have:

$$G(x)(\alpha) = \epsilon_\alpha(x) \ , \qquad G(x)(\alpha\, p\, u) = G(\delta_{\alpha p}(x))(u) \ .$$

The tuple $\langle \mathcal{R}eg(\Sigma \uplus B(A)), \delta, \epsilon \rangle$ can be seen as a deterministic automaton with input alphabet $At \times \Sigma$, and output set 2^{At}. Thanks to the above characterisation, a state x in this automaton accepts precisely the guarded string language $G(x)$—modulo the isomorphism $(At \times \Sigma)^\star \to 2^{At} \approx \mathcal{P}((At \times \Sigma)^\star \times At)$.

However, we cannot directly apply the explicit algorithm from Section 2.1, because this automaton is not finite. First, there are infinitely many KAT expressions, so that we have to restrict to those that are accessible from the expressions we want to check for equality. This is however not sufficient: we also have to quotient regular expressions w.r.t. a few simple laws [28]. This quotient is simple to implement by normalising expressions; we thus assume that expressions are normalised in the remainder of this section.

Symbolic derivatives. The input alphabet of the above automaton is exponentially large w.r.t. the number of primitive tests: $At \times \Sigma = 2^A \times \Sigma$. Therefore, the simple algorithm from Section 2.1 is not tractable in practice. Instead, we would like to use its symbolic version (Figure 5).

The output values, in $(2^{At} = 2^A \to 2)$, are also exponentially large, and are best represented symbolically, using Boolean BDDs. In fact, any test appearing in a KAT expression can be pre-compiled into a Boolean BDD: rather than working with regular expressions over $\Sigma \uplus B(A)$ we thus move to regular expressions over $\Sigma \uplus \mathrm{BDD}_A[2]$, which we call *symbolic KAT expressions*. We denote the set of such expressions by SyKAT, and we let $(\!|e|\!)$ denote the symbolic version of a KAT expression e.

Note that there is a slight discrepancy here w.r.t. Section 3: the input alphabet is $2^A \times \Sigma$ rather than just $2^{A'}$ for some A'. For the sake of simplicity, we just assume that Σ is actually of the shape $2^{\Sigma'}$; alternatively, we could work with automata whose transition functions are represented partly symbolically (for At), and partly explicitly (for Σ)—this is what we do in the implementation.

We define the symbolic derivation operations in Figure 9.

The output function, ϵ^{s}, has type $\mathrm{SyKAT} \to \mathrm{BDD}_A[2]$, it maps symbolic KAT expressions to Boolean BDD nodes. The operations used on the right-hand side of this definition are those on Boolean BDDs. The function ϵ^{s} is much more efficient than its explicit counterpart (ϵ, in Figure 8): the set of all accepted atoms is computed at once, symbolically.

The function δ^{s} has type $\mathrm{SyKAT} \to \mathrm{BDD}_{A \uplus \Sigma'}[\mathrm{SyKAT}]$. It maps symbolic KAT expressions to BDDs whose leaves are themselves symbolic KAT expressions. Again, in contrast to its explicit counterpart, δ^{s} computes all the transitions of a given expression once and for all. The operations used on the right-hand side of the definition are the following ones:

- $n \oplus m$ is defined by applying pointwise the syntactic sum operation from KAT expressions to the two BDDs n and m: $n \oplus m = \lfloor \phi \mapsto \lceil n \rceil(\phi) + \lceil m \rceil(\phi) \rfloor$;
- $n \odot x$ syntactically multiplies all leaves of the BDD n by the expression x, from the right: $n \odot x = \lfloor \phi \mapsto \lfloor n \rceil(\phi) \cdot x \rfloor$;
- $f \otimes n$ "multiplies" the Boolean BDD f with the BDD n: $f \otimes n = \lfloor \phi \mapsto \lfloor n \rceil(\phi)$ if $\lfloor f \rceil(\phi) = 1, 0$ otherwise$\rfloor$.
- $\lfloor q \mapsto 1, \_ \mapsto 0 \rfloor$ is the BDD mapping q to 1 and everything else to 0 ($q \in \Sigma = 2^{\Sigma'}$ being cast into an element of $2^{A \uplus \Sigma'}$).

By two simple inductions, one proves that for every expression $x \in \mathrm{SyKAT}$, atom $\alpha \in At$, and letter $p \in \Sigma$, we have:

$$\lceil \epsilon^{\mathrm{s}} (\!| x |\!) \rceil(\alpha) = \epsilon_\alpha(x)$$
$$\lceil \delta^{\mathrm{s}} (\!| x |\!) \rceil(\alpha p) = (\!| \delta_{\alpha p}(x) |\!)$$

(Again, we abuse notation by letting the pair αp denote an element of $2^{A \uplus \Sigma'}$.) This ensures that the symbolic deterministic automaton $\langle \mathrm{SyKAT}, \delta^{\mathrm{s}}, \epsilon^{\mathrm{s}} \rangle$ faithfully represents the previous explicit automaton, and that we can use the symbolic algorithms from Section 3.

4.2 Partial derivatives

An alternative to Brzozowski's derivatives consists in using Antimirov' *partial derivatives* [4], which generalise to KAT in a straightforward way [34]. The difference with Brzozowski's derivative is that they produce a non-deterministic automaton: states are still expressions, but the derivation function produces a set of expressions. An advantage is that we do not need to normalise expressions on the fly: the set of partial derivatives reachable from an expression is always finite.

We give directly the symbolic definition, which is very similar to the previous one.

$$\begin{aligned}
{\delta'}^{\mathrm{s}}(x{+}y) &= {\delta'}^{\mathrm{s}}(x) \mathbin{\underline{\cup}} {\delta'}^{\mathrm{s}}(y) \\
{\delta'}^{\mathrm{s}}(x{\cdot}y) &= ({\delta'}^{\mathrm{s}}(x) \boxdot y) \mathbin{\underline{\cup}} (\epsilon^{\mathrm{s}}(x) \boxtimes {\delta'}^{\mathrm{s}}(y)) \\
{\delta'}^{\mathrm{s}}(x^\star) &= {\delta'}^{\mathrm{s}}(x) \boxdot x^\star \\
{\delta'}^{\mathrm{s}}(p) &= \lfloor p \mapsto \{1\}, _ \mapsto \emptyset \rfloor \\
{\delta'}^{\mathrm{s}}(\phi) &= \emptyset
\end{aligned}$$

The differences lie in the BDD operations, whose leaves are now sets of expressions:

- $n \mathbin{\underline{\cup}} m = \lfloor \phi \mapsto \lceil n \rceil(\phi) \cup \lceil m \rceil(\phi) \rfloor$;
- $n \boxdot x = \lfloor \phi \mapsto \{x' \cdot x \mid x' \in \lceil n \rceil(\phi)\} \rfloor$;
- $f \boxtimes n = \lfloor \phi \mapsto \lceil n \rceil(\phi)$ if $\lceil f \rceil(\phi) = 1, \emptyset$ otherwise$\rfloor$.

One can finally relate partial derivatives to Brzozowski's one:

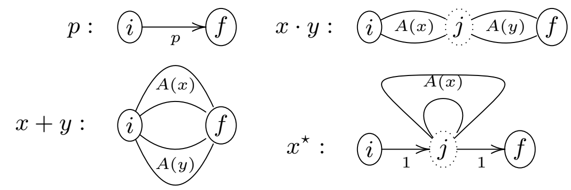
$$\mathrm{KA} \vdash \sum \lceil {\delta'}^{\mathrm{s}}(x) \rceil(\alpha p) = (\!| \delta_{\alpha p}(x) |\!) \enspace .$$

(The above Σ denotes the iterated sum of the set of partial derivatives—we do not have a syntactic equality because partial derivatives inherently exploit the fact that multiplication distributes over sums.) Using symbolic determinisation as described in Section 3.2, one can thus use the algorithm from Section 3 with Antimirov' partial derivatives.

4.3 Ilie & Yu's construction

Other automata constructions from the literature can be generalised to KAT expressions. We can for instance consider Ilie and Yu's construction [25], which produces non-deterministic automata with epsilon transitions with exactly one initial state, and one accepting state.

We consider a slightly simplified version here, where we omit a few optimisations and just proceed by induction on the expression. The four cases are depicted below: i and f are the initial and accepting states, respectively; in the concatenation and star cases, a new state j is introduced.



To adapt this construction to KAT expressions, it suffices to generalise epsilon transitions to transitions labelled by tests. In the base case for a test ϕ, we just add a transition labelled by ϕ between i and f; the two epsilon transitions needed for the star case just become transitions labelled by the constant test 1.

As expected, when starting from a symbolic KAT expression, those counterparts to epsilon transitions are labelled by Boolean BDD nodes rather than by explicit Boolean expressions.

Epsilon cycles. The most important optimisation we miss with this simplified presentation of Ilie and Yu's construction is that we should merge states that belong to cycles of epsilon transitions. An alternative to this optimisation consists in normalising first the expressions so that for all subexpressions of the shape $e^\star$, e does not contain 1, i.e., $\epsilon^{\mathrm{s}}(e) \neq 1$. Such a normalisation procedure has been proposed for plain regular expressions by Brüggemann-Klein [8]. When working with such normal forms, the automata produced by the above simplified construction (on plain regular expressions) have acyclic epsilon transitions, so that the aforementioned optimisation is unnecessary.

This normalisation procedure generalises easily to (symbolic) KAT expressions. For instance, here are typical normalisations:

$$(\phi + p)^\star \mapsto p^\star \tag{1}$$
$$(p^\star + q)^\star \mapsto (p + q)^\star \tag{2}$$
$$((1 + p)(1 + q))^\star \mapsto (p + q)^\star \tag{3}$$

We say that Symbolic KAT expressions satisfying the above property are in *strict star form*. The normalisation procedure is linear in the size of the expressions; it always produces a smaller expression. As a consequence, when deciding whether a KAT equation holds or not, it is always beneficial to put the expressions in strict star form first, independently from the considered automata construction. (See the experiments in Section 5).

According to the example (1), it might be tempting to strengthen example (3) into $((\phi + p)(\psi + q))^\star \mapsto (p + q)^\star$. Such a step is invalid, unfortunately. (The second expression accepts the guarded string $\alpha p \beta$ for all α, β, while the starting expression needs $\beta \vDash \psi$.) This example seems to show that one cannot ensure that all starred

	iterations		time		NFA states	DFA states
	`symb_equiv`	`dsf_equiv`	`symb_equiv`	`dsf_equiv`		
Antimirov ∘ ssf	6 715	3 980	0.53s	0.47s	2 704	4 142
Antimirov	7 141	4 256	0.84s	0.74s	3 039	4 442
Ilie & Yu ∘ ssf	6 985	4 209	1.77s	1.73s	4 716	4 441
Ilie & Yu	7 328	4 445	3.89s	3.83s	5 730	4 647
Brzozowski ∘ ssf	11 952	6 525	6.88s	4.67s	-	6 684
Brzozowski	19 781	10 080	43.00s	30.00s	-	10 265

Table 1. Checking random saturated pairs of expressions.

subexpressions are mapped to 0 by ϵ^{s}. As a consequence we cannot assume that test-labelled transitions generated by Ilie and Yu's construction form an acyclic graph in general.

4.4 Test-labelled transitions removal

The above construction produces symbolic NFA with test-labelled transitions, which have to be eliminated in order to apply the algorithms from Section 3. Other constructions from the literature produce automata with epsilon transitions and can be adapted to KAT using test-labelled transitions. A generic procedure for eliminating such transitions is thus desirable.

The usual technique with plain automata consists in computing the reflexive transitive closure of epsilon transitions, to precompose the other transitions with the resulting relation, and declare a state as accepting in the new automaton whenever it can reach an accepting state through this reflexive-transitive closure.

More formally, let us recall Kozen's matricial representation of non-deterministic automaton with epsilon transitions [26], as tuples $\langle n, u, J, N, v\rangle$, where u is a $(1, n)$ 01-matrix denoting the initial states, J is a (n, n) 01-valued matrix denoting the epsilon transitions, N is a (n, n) matrix representing the other transitions (with entries sets of letters in Σ), and v is a $(n, 1)$ 01-matrix encoding the accepting states.

The language accepted by such an automaton can be represented by the following matricial product, using Kleene star on matrices:

$$u \cdot (J + N)^\star \cdot v \ .$$

Thanks to the algebraic law $(a+b)^\star = a^\star \cdot (b \cdot a^\star)^\star$, which is valid in any Kleene algebra, we get

$$\mathrm{KA} \vdash u \cdot (J + N)^\star \cdot v = u \cdot (J^\star N)^\star \cdot (J^\star v) \ .$$

We finally check that $\langle n, u, 0, J^\star N, J^\star v\rangle$ represents a non-deterministic automaton without epsilon transitions. This is how Kozen validates epsilon elimination for plain automata, algebraically [26].

The same can be done here for KAT by noticing that tests (or Boolean BDD nodes) form a Kleene algebra with a degenerate star operation: the constant-to-1 function. One can thus generalise the above reasoning to the case where J is a tests-valued matrix rather than a 01-matrix.

The iteration $J^\star$ of such a matrix can be computed using standard shortest-path algorithms [21], on top of the efficient semiring of Boolean BDD nodes. The resulting automaton has the expected type:

- there is a transition labelled by αp between i and j if there exists a k such that $\alpha \vDash (J^\star)_{i,k}$ and $p \in N_{k,j}$. (The corresponding non-deterministic symbolic transition function can be computed efficiently using appropriate BDD functions.)
- The output value of a state i is the Boolean BDD node obtained by taking the disjunction of all the $(J^\star)_{i,j}$ such that j is an accepting state (i.e., just $(J^\star)_{(i,f)}$ when using Ilie and Yu's construction).

5. Experiments

We implemented all presented algorithms in OCaml; the corresponding library is available online, together with an applet allowing to trace them on user-provided examples [35].

Symbolic KAT expressions are hash-consed, which allows us to represent sets of expressions using Patricia trees (e.g., for Antimirov' partial derivates). Expressions are also normalised using smart constructors: sums associated to the left, sorted, and without duplicates; products are associated to the left; consecutive tests are merged; units are cancelled as much as possible. For Antimirov' construction and for Ilie and Yu's construction, the produced symbolic NFA are memoised once and for all, and reindexed so that their states are just natural numbers. This allows us to use bit-vectors to represent sets produced during determinisation. The queue `todo` used for storing the pairs to process is a FIFO queue, so that the automata are explored in a breadth-first manner.

We performed a few experiments to compare the presented algorithms and constructions. We generated random KAT expressions over two sets of seven primitive tests and seven atomic elements, with seventy connectives, and excluding explicit occurrences of the constants 0 and 1. A hundred pairs of random expressions were checked for equality after being saturated by adding the constant $\Sigma^\star$ on both sides. (A difficulty here is that random pairs of expressions are almost always distinguished by a very short guarded string, which is found almost immediately thanks to the breadth-first strategy, independently from the size of the expressions and from the up-to techniques at work. Instead, we would like to evaluate the algorithms based on their running time on more interesting pairs, where the expressions are either equivalent or distinguished only by long guarded strings. By saturating the expressions with the constant $\Sigma^\star$, we artificially make the expressions equivalent. Moreover, looking at an execution of the presented algorithms on such saturated pairs, what happens is that the output test (line 10 on Figure 1) always succeeds, so that the algorithms stop only once the whole automata have been explored and a bisimulation has been found. Moreover, an analysis of the various automata constructions shows that the automata constructed for an expression p are very similar to the automata constructed for the expression $p + \Sigma^\star$: exploring the latter is as hard as exploring the former.)

The results are displayed in Table 1: for each construction and for each of the two symbolic algorithms, we give the total number of iterations (i.e., the number of times we execute line 10 in Figure 5), and the global running time[2]. Each construction is associated to two lines, depending on whether we first put expressions in strict star form or not. We additionally provide the total number of NFA states generated by Antimirov' and Ilie and Yu's constructions, as well as the total number of DFA states generated for the three constructions.

One can notice than Antimirov' partial derivatives provide the fastest algorithms. Ilie and Yu's construction yield approximately

[2] Theses experiments were performed on a MacBook Pro, OS X 10.9.5, 2,4GHz Intel Core i7, 4Go 1333MHz DDR3, OCaml 4.02.1

the same number of iterations as Antimirov' partial derivatives, but require more time: computing the transitive closure for epsilon removal is a costly operation. Brzozowski's construction gives poor results both in terms of time and iterations: the produced automata are larger, and more difficult to compute.

Concerning the equivalence algorithm, one notices that using disjoint set forests significantly reduces the number of iterations. There is almost no difference in the running times with the first two constructions, because most of the time is spent in constructing the automata rather than checking them for equivalence. This is no longer true with Brzozowski's construction, for which the automata are sufficiently big to observe a difference.

6. Directions for future work

The equational theory of KAT is PSPACE-complete, but none of the presented algorithms are PSPACE (just because of the use BDDs, but also because the bisimulation candidate, which has to be stored, can be exponentially large). Experiments however suggest that they can be useful in practice: the symbolic DFA produced by the various constructions proposed in this paper tend to be of reasonable size. Quantifying this empirical observation in a formal way seems extremely difficult.

A natural extension of this work would be to apply the proposed algorithms to KAT+B! [19] and NetKAT [2], two extensions of KAT with important applications in verification: while programs with mutable tests in the former case, and network programming in the later case.

KAT+B! has a EXPSPACE-complete equational theory, and its structure makes explicit algorithms completely useless. Designing symbolic algorithms for KAT+B! seems challenging.

NetKAT remains PSPACE-complete, and Foster et al. propose in the present volume a coalgebraic decision procedure relying on a variation of Antimirov' derivatives [17]. To get a practical algorithm, they represent automata transitions using sparse matrices, and they exploit some form of symbolic treatment by using what they call "bases". KAT can be encoded into NetKAT, so that their algorithm could be used for KAT. This encoding is however not streamlined, and it is non-trivial to understand the behaviour of their algorithm on the resulting instances. Conversely, adapting the algorithms presented in the present paper to cope with NetKAT seems feasible, although not straightforward. Concerning the symbolic treatment of automata, our use of BDDs seems more powerful and less ad-hoc than their use of bases, but the precise relationship remains unclear, and we leave its formal analysis for future work.

Moving away from KAT specificities, we leave open the question of the complexity of our symbolic variant of Hopcroft and Karp's algorithm (Figure 7). Tarjan proved that their algorithm is almost linear in amortised time complexity, and he made a list of heuristics for linking and path compression schemes that lead to that complexity [40]; together with Goel, Khanna and Larkin, he recently showed that this complexity is still reached (asymptotically) with randomized linking [18]. A similar study for the symbolic counterpart we propose here remains to be done.

Acknowledgments

We are grateful to the anonymous referees who provided thorough and detailed reviews, and in particular to the one who noticed the relationship between the present work and Rémy's type inference algorithm for row types.

References

[1] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *Proc. TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 158–174. Springer Verlag, 2010.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: semantic foundations for networks. In *Proc. POPL*, pages 113–126. ACM, 2014.

[3] A. Angus and D. Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, CS Dpt., Cornell University, July 2001.

[4] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[5] S. L. Bloom, Z. Ésik, and G. Stefanescu. Notes on equational theories of relations. *Algebra Universalis*, 33(1):98–126, 1995.

[6] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In *Proc. POPL*, pages 457–468. ACM, 2013.

[7] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer Verlag, 2004.

[8] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[10] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[11] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[12] E. Cohen, D. Kozen, and F. Smith. The complexity of Kleene algebra with tests. Technical Report TR96-1598, CS Dpt., Cornell University, 1996.

[13] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *POPL*, pages 541–554. ACM, 2014.

[14] L. Doyen and J.-F. Raskin. Antichain Algorithms for Finite Automata. In *Proc. TACAS*, volume 6015 of *Lecture Notes in Computer Science*. Springer Verlag, 2010.

[15] Z. Ésik and L. Bernátsky. Equational properties of Kleene algebras of relations with conversion. *Theoretical Computer Science*, 137(2):237–251, 1995.

[16] J.-C. Filliâtre and S. Conchon. Type-safe modular hash-consing. In *ML*, pages 12–19. ACM, 2006.

[17] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for NetKAT. In *Proc. POPL*. ACM, 2015.

[18] A. Goel, S. Khanna, D. Larkin, and R. E. Tarjan. Disjoint set union with randomized linking. In *Proc. SODA*, pages 1005–1017. SIAM, 2014.

[19] N. B. B. Grathwohl, D. Kozen, and K. Mamouras. KAT + B! In *Proc. CSL-LICS*. ACM, July 2014.

[20] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer Verlag, 1995.

[21] P. Höfner and B. Möller. Dijkstra, Floyd and Warshall meet Kleene. *Formal Aspects of Computing*, 24(4-6):459–476, 2012.

[22] J. E. Hopcroft. An $n\log n$ algorithm for minimizing states in a finite automaton. Technical report, Stanford University, 1971.

[23] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 114, Cornell University, December 1971.

[24] G. Huet. *Résolution d'équations dans les langages d'ordre 1,2, ... ,ω*. PhD thesis, Université Paris VII, 1976. Thèse d'État.

[25] L. Ilie and S. Yu. Follow automata. *Information and Computation*, 186(1):140–162, 2003.

[26] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[27] D. Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[28] D. Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical report, CIS, Cornell University, March 2008.

[29] D. Kozen and M.-C. Patron. Certification of compiler optimizations using Kleene algebra with tests. In *Proc. CL2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 568–582. Springer Verlag, 2000.

[30] D. Kozen and F. Smith. Kleene algebra with tests: Completeness and decidability. In *Proc. CSL*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259. Springer Verlag, September 1996.

[31] E. F. Moore. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Mathematical Studies*, 34:129–153, 1956.

[32] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[33] F. Pottier and D. Rémy. *Advanced Topics in Types and Programming Languages*, chapter The Essence of ML Type Inference. MIT Press, 2004.

[34] D. Pous. Kleene Algebra with Tests and Coq tools for while programs. In *Proc. ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 180–196. Springer Verlag, 2013.

[35] D. Pous. Web appendix to this paper, with Ocaml implementation of the proposed algorithms, 2014. `http://perso.ens-lyon.fr/damien.pous/symbolickat`.

[36] D. Pous and D. Sangiorgi. *Advanced Topics in Bisimulation and Coinduction*, chapter about "Enhancements of the coinductive proof method". Cambridge University Press, 2011.

[37] D. Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels*. PhD thesis, Université Paris VII, 1990. Thèse de doctorat.

[38] D. Rémy. Extension of ML type system with a sorted equational theory on types, 1992. Research Report 1766.

[39] D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8:447–479, 1998.

[40] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[41] M. Veanes. Applications of symbolic finite automata. In *CIAA*, volume 7982 of *Lecture Notes in Computer Science*, pages 16–23. Springer Verlag, 2013.

[42] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer Verlag, 2006.