Towards Distributed Software Model-Checking using Decision Diagrams[*] (extended version with annexes)

Maximilien Colange[1], Souheib Baarir[2], Fabrice Kordon[1], and Yann Thierry-Mieg[1]

[1] LIP6, CNRS UMR 7606, Université P. & M. Curie – Paris 6
4, place Jussieu, F-75252 Paris Cedex 05, France
[2] LIP6, CNRS UMR 7606 and Université Paris Ouest Nanterre La Défense
200, avenue de la République, F-92001 Nanterre Cedex, France
first.last@lip6.fr

Abstract. Symbolic data structures such as Decision Diagrams have proved successful for model-checking. For high-level specifications such as those used in programming languages, especially when manipulating pointers or arrays, building and evaluating the transition is a challenging problem that limits wider applicability of symbolic methods.

We propose a new symbolic algorithm, *EquivSplit*, allowing an efficient and fully symbolic manipulation of transition relations on Data Decision Diagrams. It allows to work with equivalence classes of states rather than individual states. Experimental evidence on the concurrent software oriented benchmark BEEM shows that this approach is competitive.

1 Introduction

Model-checking of concurrent software faces state space explosion. To address this issue, many algorithms and data structures have been proposed, one of the most successful being symbolic shared data structures such as Binary Decision Diagrams (BDD).

While BDD allow in many cases to cope with very large state spaces, expressing algorithms symbolically to take full advantage of the data structure is tricky. Symbolic evaluation algorithms that are aware of the data structure itself such as saturation-style algorithms [6, 10] can be orders of magnitude better than naive evaluation in a breadth-first search manner.

The transition relation of a system of k boolean variables, can be seen as a function $\mathbb{B}^k \mapsto 2^{\mathbb{B}^k}$ and is usually built and stored as a second decision diagram N, with two variables "before" and "after" for each variable of the system. A specific operation between any subset of the state space S encoded as a decision diagram and the transition relation N yields a decision diagram $S' = N(S)$ representing immediate successors of S.

Let us define statements as (sequences of) assignments of expressions to variables. The support of a statement is the set of variables it reads or writes to. This notion of

[*] This work has been supported by a grant from the Délégation Générale pour l'Armement and by the project ImpRo/ANR-2010-BLAN-0317.

locality is heavily exploited, to limit the representation of transitions to the effect they have on variables of their support. For each transition with k' Boolean support variables, worst case representation size is $2^{k'}$. The symbolic approach was successfully applied to Boolean gate logic where encoding these $\mathbb{B}^{k'} \mapsto 2^{\mathbb{B}^{k'}}$ transition matrices is feasible.

But because classical approaches compute potential to potential $\mathbb{B}^{k'} \mapsto 2^{\mathbb{B}^{k'}}$ transition matrices, a larger support for transitions means exponential growth of the worst case complexity in representation size. It also severely limits the possibilities of saturation-based techniques as their efficiency relies in clusters based on the support of transitions. Hence, a worst case for classical symbolic approaches is when the support of transitions includes all variables.

Moreover, when the input specification includes array or pointer manipulation, any static analysis of statements will necessarily yield pessimistic support assumptions. For instance, a non-constant array access such as `t[i]` *may* depend on the variable `t[0]`. In classical approaches, pessimistic assumptions must include all elements of the array t in the support. Such expressions are commonly encountered in modeling languages such as Promela or Divine [11, 2].

We propose in this paper to perform a dynamic analysis of such statements as they are being resolved, allowing to discover more locality in the remaining effects as expressions are partially evaluated. This can avoid the problems induced by transitions with a large syntactic support by only performing the computations that are *really necessary*. Our algorithm exploits locality to optimize its evaluation, as the support of expressions may vary as the evaluation progresses.

In the dynamic case, when evaluating `t[i]`, as soon as the value of the index expression `i` has been reduced to a constant, pessimistic assumptions can be forgotten and the support is reduced to the effective cell of the array that is the target of the assignment.

To have efficient symbolic computations of these statements, we define an equivalence relation over states with respect to the value of an expression; this induces equivalence classes that can be built dynamically and manipulated symbolically. Intuitively, if efficient manipulation of equivalence classes is possible, then the computation complexity can be proportional to the number of such equivalence classes rather than to the number of actual states.

We define in this paper a new decision diagram based operation, *EquivSplit*, that allows to efficiently compute and manipulate such equivalence classes, in a way compatible with the decision diagram encoding of states. Given a syntax tree e for an arbitrary expression, and a set of states S encoded as a decision diagram, we provide an incremental and on the fly algorithm to efficiently compute a partition of $S = S_0 \uplus S_1 \uplus \ldots$ where all states in a S_i agree on the value of e, and no two distinct S_i, S_j agree on the value of e.

Outline. We first introduce notations for expressions and their (partial) evaluation. We then recall the definition of Data Decision Diagrams (DDD)[9], as the type of integer valued decision diagrams we use in our implementation. We then explain the *EquivSplit* algorithm and how it is used to evaluate and resolve expressions on sets of values stored as DDD. To assess the applicability of our approach in practice, we study in section 5 the efficiency of our approach for Divine models taken from a standard benchmark (BEEM) and compare it to other symbolic approaches.

2 Expressions

We first define in 2.1 some concepts and introduce notations that will be used throughout the paper. The abstract level of these definitions guarantees independence from any concrete syntax. We give flesh to these definitions with more concrete examples in 2.2.

2.1 Definitions and notations

Let Σ be a signature, that is a set of symbols of finite arity. We inductively define the set `Expr` of Σ-*expressions* as $\phi \in$ `Expr` if and only if:

- $\phi \in \Sigma$ of arity 0,
- or $\phi = s(\phi_1, \ldots, \phi_k)$ where $s \in \Sigma$ is of arity k and $\phi_1, \ldots, \phi_k \in Expr$ (ϕ_i is called a sub-expression).

Let D be a domain for expressions. We assume that D is embedded in Σ, so that every element of the domain can be referred to syntactically.

Definition 1. *An* interpretation *I is a function that associates to every symbol $s \in \Sigma$ of arity $k > 0$ a (possibly partial) function $I(s) : D^k \mapsto D$, and that maps each symbol of arity 0 to its corresponding element of D.*

Intuitively, this formalism captures most programming languages, with pointers and pointer arithmetic. From now on, we assume that there is a finite subset X in D, called *addresses*. The set of addresses X being finite, we note $X = \{x_1, \ldots, x_{|X|}\}$. We assume Σ contains a special symbol δ of arity 1, that allows to access a memory slot given its address. Note that a variable is just a symbolic name for an address. Thus, $I(\delta)$ represents the content of the memory that varies as the program runs. Since we focus on the evolution of the content of the memory, all the interpretations considered from now on are equal for the other symbols (i.e. the operational semantics for the symbols of the language is known and fixed). Let $\mu = I(\delta)$ designate a *valuation*, i.e. the state of the memory. μ is seen as a (partial, when not all memory contents are known) function from X into D. Since all other symbols have a fixed interpretation, an interpretation I can be described by simply providing μ. Furthermore, all symbols interpretations must be complete functions (only the valuation is allowed to be a partial function). Partial interpretations can be completed by adding a special element to D and mapping the undefined domain onto this special element. This special element corresponds to an error or an undefined behavior. Note that the interpretations of all symbols must take into account this new special element.

Definition 2. *Given an interpretation I, an expression $\phi = s(\phi_1, \ldots, \phi_k)$ ($k \geq 0$)* evaluates *or* reduces *to another expression $eval(I, \phi)$ as follows:*

$$eval(I,\phi) = \begin{cases} I(s) \in D & \text{if } s \text{ is a symbol of arity } 0 \\ I(s)(eval(I,\phi_1), \ldots, eval(I,\phi_k)) \in D & \text{if } eval(I,\phi_i) \in D \text{ for all } i \text{ and } I(s) \text{ is defined at this point} \\ s(eval(I,\phi_1), \ldots, eval(I,\phi_k)) & \text{otherwise.} \end{cases}$$

If $eval(I, \phi) \in D$, the evaluation is complete.

Notation. We will now abusively denote the evaluation $eval(I,\phi)$ where $I(\delta) = \mu$ by $eval(\mu,\phi)$. If ψ is a (possibly nested) sub-expression of ϕ, $\phi[\psi \leftarrow \theta]$ denotes the expression obtained by substituting the expression θ to ψ in ϕ. Given a valuation μ and a subset of addresses $Y \subseteq X$, $\mu_{|Y}$ denotes the restriction of μ to Y. With these notations, we have, for any variable x, any valuation μ where x is defined, and any expression ϕ: $\phi[\delta(x) \leftarrow \mu(x)] = eval(\mu_{|\{x\}},\phi)$

We now define an equivalence relation on valuations with respect to the evaluation of an expression. In Section 4 this equivalence relation is a key notion, allowing efficient evaluation of expressions on sets of valuations.

Definition 3. *Given a subset Y of X and an expression ϕ, for all valuations μ,μ' we define the equivalence relation $\sim_{\phi}^{Y}$ as follows:*

$$\mu \sim_{\phi}^{Y} \mu' \Leftrightarrow eval(\mu_{|Y},\phi) = eval(\mu'_{|Y},\phi)$$

A trivial case of this equivalence is valuations $\mu \neq \mu'$, that are equal on Y.

2.2 Examples of Expressions

To help in visualizing these definitions, let us use as an example a language supporting a C-like syntax. We give concrete examples here for each element defined abstractly above. We consider a language supporting integers and their manipulation operators (arithmetic +, -, * ... as well as bitwise operations <<,>>,...). The set of considered operators are part of the signature Σ. The domain D is thus integers. The Σ-expressions are built by syntactic combinations of operators, and the literals 0 or 1 are also (terminal) expressions (as D is embedded in Σ).

Then, by definition 1, we must provide an interpretation function I that gives the semantics of all the operators which are used in expressions. The interpretation function works with constants; for our example we should provide the integer output value for each of the binary operators given two integers.

Consider now variables of the program "a,b,c". They are seen as symbolic names and mapped to integers (memory addresses), for instance $0,1,2$. The special operator δ allows to read the value of such a variable, hence the expression a is interpreted as $\delta(0)$. We add the notion of array of fixed size tab, and access to a cell of an array using $tab[]$. Again tab is a symbolic name for a variable mapped to an integer, for instance 3 that is the first memory slot occupied by the array. Then $tab[e]$ where e is an arbitrary expression is a syntactic sugar for $\delta(3+e)$.

All operators should have complete interpretations: a/b must also be defined when $b = 0$. For this purpose, one or more special constants can be introduced. For a given language manipulating finite types, the definition of the interpretation of most symbols is usually straightforward. We consider that the interpretation of all symbols except δ is fixed throughout the computations. In other words we distinguish the code (all other symbols from the signature) from the data, represented by $I(\delta)$, that may vary as the computation progresses.

Definition 2 formalizes partial evaluation of expressions given an interpretation function. For instance, suppose μ only gives the content of memory slot 0, say $\mu(0) = 12$.

Let $\phi = add(\delta(0), \delta(1))$ (usually noted $a + b$). Then $eval(\mu, \phi) = add(eval(\mu, \delta(0)), eval(\mu, \delta(1)))$. We have $eval(\mu, \delta(0)) = I(\delta)(0) = \mu(0) = 12$. However, because μ is not defined for address 1, $eval(\mu, \delta(1)) = \delta(1)$. Hence, $eval(\mu, \phi) = add(12, \delta(1))$ (noted $12 + b$).

As an example for Definition 3, any two μ, μ' such that $eval(\mu, a + b) = eval(\mu', a + b)$ are equivalent. For instance, if both a and b are in Y, $\mu = (a \leftarrow 0, b \leftarrow 1), \mu' = (a \leftarrow 1, b \leftarrow 0)$ are equivalent. If only a is in Y, μ and μ' are not equivalent, since one yields expression $0 + b$ while the other yields $1 + b$.

3 Data Decision Diagrams (DDD) [9]

Let us now briefly recall important concepts of decision diagrams. The algorithm presented in this paper is valid for any type of shared decision diagram, such as BDD. However, to more closely match our definition of expressions, we will consider here Data Decision Diagrams, where the domain of variables is D rather than $\mathbb{B}$. This provides a natural representation for a set of valuations as a DDD.

Shared Decision Diagrams (DD) are a data structure to compactly represent sets. There are many variants of decision diagrams used for model-checking, but they all rely on the same underlying principles: nodes of the decision tree are unique in memory thanks to a canonical representation; the number of paths through the diagram (states) can be exponential in the representation size (nodes in the DD); equality of two sets can be tested in constant time; using caches most operations manipulating a DD are polynomial in the representation size; the effectiveness of the encoding strongly depends on the chosen variable ordering [7].

In this paper we rely on Data Decision Diagrams (DDD, defined in [9]), which extend classical BDD in two respects: *1)* variables are considered to have an integer domain instead of a Boolean one, and, *2)* operations over DDD are encoded using homomorphisms instead of the usual fashion where another decision diagram with two variables per variable of the state signature is used.

A DDD is a data structure for representing a set of sequences of assignments of the form $x_1 := v_1; x_2 := v_2; \ldots; x_n := v_n$, where x_i are variables and v_i are values in D. We assume a total order on variables such that all variables are always encountered in the same order in an assignment sequence. The usual DDD definition makes weaker assumptions on variable ordering, but these are out of the scope of this paper (see [9]).

We define the terminal **1** to represent the empty assignment sequence, that terminates any valid sequence, and **0** to represent the empty set of assignment sequences.

Definition 4 (DDD). *Let X be a set of variables ranging over domain D. The set $\mathbb{D}$ of DDD is defined inductively by:*
$\delta \in \mathbb{D}$ *if either* $\delta \in \{\mathbf{0}, \mathbf{1}\}$ *or* $\delta = \langle x, \alpha \rangle$ *with* $x \in X$, *and* $\alpha : D \to \mathbb{D}$ *is a mapping where only a finite subset of D maps to other DDD than* **0**.

By convention, edges that map to the DDD **0** *are not represented.*

For instance, consider the DDD shown in figure 1. Each path in the DDD corresponds to a sequence of assignments. In this work, we use DDD to represent valuations of the memory, thus each assignment sequence represents a memory state.

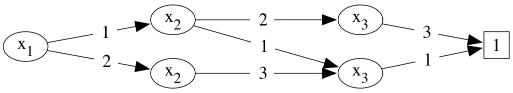


Fig. 1. This DDD with domain $D = \mathbb{N}$ represents the set of sequences of assignments: $\{(x_1 := 2; x_2 := 3; x_3 := 1;), (x_1 := 1; x_2 := 1; x_3 := 1;), (x_1 := 1; x_2 := 2; x_3 := 3;)\}$.

Operations and Homomorphisms. DDD support standard set operations: $\cup$, $\cap$, $\setminus$. The semantics of these operations are based on the sets of assignment sequences that the DDD represent.

Basic and inductive homomorphisms are also introduced to define application specific operations. A detailed description of DDD homomorphisms can be found in [9].

Since in this paper we define new symbolic operations that are not specific to DDD, we omit further details on homomorphisms. From the implementation point of view, all operations we define are embedded in homomorphisms. This allows the software library to enable automatic rewritings that yield much better performances, such as the saturation algorithm [10].

4 Evaluating expressions on DDD

In practice, a system's state is a valuation of the state variables, and the behavior of the system is described with expressions. When treating such a system using DDD arises the need to evaluate an expression over a *set* of valuations.

More precisely, given an expression ϕ and a set of valuations V, one needs to compute all the evaluations of ϕ by the valuations in V. To achieve this goal efficiently, we rely on equivalence relation $\sim_\phi^X$ of definition 3.

Recall that the size of a DDD is often logarithmic in the size of the represented set. The naive approach considers each valuation separately, ending up with a complexity linear in the size of the input set. An efficient solution to this problem should use functions that manipulate the nodes of the data structure representation, so that thanks to caches, the complexity remains proportional to the encoding size.

We propose an algorithm, *EquivSplit*, that partitions a set of valuations (given as a DDD) into equivalence classes with respect to $\sim_\phi^X$. It visits variables in the order given by the DDD, and progressively evaluates the expression. Hence it must work with partial valuations and partially evaluated expressions.

We first define in section 4.1 the notion of dependency on an address, and how to resolve such dependencies to ensure proper recursion. We then present our algorithm in a restricted case to help comprehension in section 4.2. It is extended to the general case, by introducing another function *SolveSub* in section 4.3. The correction and the complexity of these functions are discussed in section 4.4.

4.1 Support of expressions

The support of an expression is the set of memory addresses necessary to completely evaluate this expression. Conversely, an expression does not depend on an address if its content does not affect its evaluation. We formally define these notions, and then explain how to partially evaluate an expression until dependencies on a given address are eliminated.

Definition 5. *An expression* ϕ does not depend *on an address x if and only if:*

$$\forall \mu, \mu' \in D^X, \mu_{|X \setminus \{x\}} = \mu'_{|X \setminus \{x\}} \implies eval(\mu, \phi) = eval(\mu', \phi)$$

The support *of an expression ϕ is the set of addresses on which ϕ depends.*
An expression that depends on no variable is said to be constant*.*

Lemma 1. *If ϕ is an expression that depends on x, then there exist a sub-expression $\delta(\psi)$ of ϕ and a valuation μ such that $eval(\mu, \psi) = x$.*
ψ is called an x-expression of ϕ.

Proof. We prove the contraposition. Let ϕ be an expression such that for all its sub-expressions of the form $\delta(\psi)$, there is no valuation μ such that $eval(\mu, \psi) = x$. Let now μ and μ' be two valuations that agree on $X \setminus \{x\}$. By structural induction on ϕ, $eval(\mu, \phi)$ (resp. $eval(\mu', \phi)$) does not depend on the value of $\mu(x)$ (resp. $\mu'(x)$). Hence, $eval(\mu, \phi) = eval(\mu', \phi)$ and we conclude that ϕ does not depend on x. □

Lemma 2. *If ϕ contains no nested δ operator, then x is not in the support of $\psi = eval(\mu_{|\{x\}}, \phi)$ for all valuations μ and addresses x. The converse is not true.*

Proof. We also prove this lemma by contraposition. Assume there exists a μ such that ψ has an x-expression ψ'. There exists an x-expression ϕ' of ϕ such that $eval(\mu_{|\{x\}}, \phi') = \psi'$. If ψ' were constant, then, according to definition 2, ψ' would be in D, and since it is an x-expression, ψ' would necessarily be equal to x. Thus, according to definition 2, $\delta(\phi')$ would be replaced by $\mu(\psi') = \mu(x)$ in ψ, so that ψ' would not be a sub-expression of ψ. This is contradictory, and proves that ψ' is not constant.
ψ' thus depends on at least an address $y \in X$, and, according to lemma 1, contains an occurrence of δ. It implies that ϕ' also contains an occurrence of δ, showing that ϕ contains nested δ operators.

Let $+$ denote any binary symbol in Σ. If $\phi = \delta(\delta(x) + \delta(y))$ and $\mu(x) + \mu(y) = x$, then $\delta(\mu(x) + \delta(y))$ still depends on x. This counter-example to the converse implication can be extended to a symbol of any arity $n \geq 2$, in case Σ contains no binary symbol. □

When there are nested δ operators, Lemma 2 states that substituting the content of an address x in ϕ, as section 4.2 naively does, may not completely remove the dependence on x. However, we can reduce this general case to the previous one by recursively solving nested x-expressions. This procedure terminates since each iteration strictly reduces the number of nested δ operators. This is discussed in section 4.3 and the correctness in section 4.4.

4.2 Without nested δ operators

The algorithm *EquivSplit* is shown in Algo. 1. It builds equivalence classes for $\sim_\phi^X$ dynamically based on successive substitution, refinement and merge steps on a partition of the input set. At step i:

– the substitution step uses the partition according to all possible contents of current address x_i (directly provided by the DDD encoding of valuations), to evaluate ϕ with each of these values;

– the refinement step refines the partition by recursively evaluating the reduced expressions over addresses $x_{i+1},\ldots,x_{|X|}$;

– the merge step merges cells of the partition that lead to the same reduced expression over addresses $x_i,\ldots,x_{|X|}$.

At each step i, the goal becomes to remove any dependencies on x_i from the expression ϕ, allowing recursion over $x_{i+1},\ldots,x_{|X|}$.

This algorithm is mutually recursive with *SolveSub* invoked on line 8. To help comprehension, we first consider the restricted case where no nested δ occur. In such a case, $\mathit{SolveSub}(\phi,V,i)$ always returns the singleton $\{(\phi,V)\}$. Hence, we study in this section the Algo. 1 independently from the algorithm of *SolveSub* presented in section 4.3.

From a programming language point of view, forbidding nested δ operators means that all addresses are known at compile time, and that no arithmetic on pointers occurs. By lemma 2, this restriction implies that if ϕ is an expression, x an address and μ a valuation, once ϕ is reduced with $\mu(x)$, it no longer depends on x.

Algorithm 1: EquivSplit(ϕ,V,i)

```
Input: $\phi$ an expression that does not depend on $x_1,\ldots,x_{i-1}$
Input: $V$ a finite set of valuations
Input: $i$ an integer between 1 and $|X|+1$
Output: a set of pairs $\{(\phi_1,c_1),\ldots,(\phi_n,c_n)\}$ such that $c_1,\ldots,c_n$ are the equivalence
        classes of $\sim_\phi^{\{x_i,\ldots x_n\}}$ over $V$, and for each $1\leq j\leq n$, $\phi_j = eval(\mu_{|\{x_i,\ldots x_n\}},\phi)$ for
        any $\mu\in c_j$.
1  if $\phi$ is constant then
2      return $\{(V,\phi)\}$
3  else
4      $map < Expr, 2^V > res$
5      let $\alpha_d = \{\mu\in V | \mu(x_i) = d\}$ for $d\in D$
6      foreach $\alpha_d \neq \emptyset$ do
           // Substitution
7          $\theta = \phi[\delta(x_i)\leftarrow d]$
           // to remove nested δ operators
8          for $(\psi,c)\in SolveSub(\theta,\alpha_d,i)$ do
               // Refinement
9              for $(\psi',c')\in EquivSplit(\psi,c,i+1)$ do
                   // Merge
10                 $res[\psi'] = res[\psi']\cup c'$
11 return $res$
```

The base case of the recursion is when ϕ is constant, hence $\sim_\phi^{Y_i}$ has a single equivalence class V (lines 1-2). If $i=|X|+1$, by the precondition on the input ϕ, ϕ is constant.

The sets $(\alpha_d)_{d\in D}$ partition V into equivalence classes with respect to the value d of x_i (lines 5-6). Note that the symbolic encoding of valuations as DDD naturally provides this partition.

To each class α_d, we associate a reduced expression θ by replacing in ϕ variable x_i by its value d (line 7). Under our simplifying assumption, θ no longer depends on x_i, and $\mathit{SolveSub}(\theta,\alpha_d,i) = \{(\theta,\alpha_d)\}$. Thus, the loop on line 8 is reduced to a single call to line 9, that becomes: "$(\psi',c') \in \mathit{EquivSplit}(\theta,\alpha_d,i+1)$".

The loop on line 9 refines the partition element α_d (c in the general case) by recursively evaluating θ (ψ in the general case) on subsequent addresses $(x_{i+1},\ldots,x_{|X|})$. Since elements from different α_d's may yield the same final value for ϕ, line 10 merges them into the final partition into equivalence classes for $\sim_{\phi}^{\{x_i,\ldots,x_{|X|}\}}$.

Invoking *EquivSplit(*$\phi,V,0$*)* returns the equivalence classes of elements in V with respect to $\sim_{\phi}^{X}$.

4.3 With nested δ operators

We now extend our algorithm to the general case. The precondition on the input ϕ for Algo. 1 is that ϕ does not depend on $x_1,\ldots,x_{i-1}$. Hence, recursion on line 9 requires that ψ does not depend on $x_1,\ldots,x_i$. The algorithm *SolveSub* addresses this problem by reducing x_i-expressions in θ by looking ahead the values of subsequent addresses $x_{i+1},\ldots,x_{|X|}$. Lemma 2 shows that with no nested δ, looking zero addresses ahead suffices to eliminate the dependencies, and falls back to the case of 4.2. The algorithm *SolveSub* performs this reduction using the look-ahead, and returns a set of pairs $\{(\phi_1,c_1),\ldots,(\phi_n,c_n)\}$ such that c_j are sets of valuations that agree on a look-ahead reduction ϕ_j of θ and that do not depend on x_i.

SolveSub computes in *res* a partition of V, and associates to each cell a simplified expression obtained by partially resolving ϕ, until all dependencies on x_i are removed. tmp is initialized as a single cell associated to ϕ (line 3). At each step of the while loop (line 4-5), an element (ψ,c) of tmp is treated. If the current expression ψ does not depend on x_i, the pair is moved to *res* (lines 11-12). Otherwise, we let θ be an x_i-expression of ψ (line 7). Recall, by lemma 1, that such a θ exists and has less nested δ operators than ψ. Any x-expression can be chosen and will lead to a correct result, hence the algorithm has some latitude at this point. Heuristically, to favor merging of partially resolved expressions, it is desirable to first treat x-expressions with a small co-domain (e.g. solve boolean sub-expressions first). Note that this is only possible with some additional knowledge of the signature's interpretation.

Recursion by invoking *EquivSplit* with θ (line 8) refines the cell c according to the value of θ. To each of these refined cells is associated the reduction of ψ obtained by substituting θ by its value (line 9). They are then added to tmp that merges the cells according to the reduced expression ψ' (line 11).

4.4 Correctness and complexity

Sketch of the proof of correctness. We give here some intuition about the correctness of both algorithms.

```
Algorithm 2: SolveSub($\phi$,$V$,$i$)
   Input: $\phi$ an expression that does not depend on $x_1,\ldots,x_{i-1}$
   Input: $V$ a set of valuations that all agree on the value $d$ of $x_i$
   Input: $i$ an integer between 1 and $|X|$
   Output: a set of pairs $\{(\phi_1,c_1),\ldots,(\phi_n,c_n)\}$ such that $c_1,\ldots,c_n$ is a partition of $V$, and
           for each $1 \leq j \leq n$, $\phi_j$ is a reduced expression obtained by removing all
           dependencies on $x_i$ from $\phi$, and all valuations in $c_j$ agree on this reduction $\phi_j$
1  $map < Expr, 2^V > res$
2  $map < Expr, 2^V > tmp$
3  $tmp[\phi] = V$
4  while $tmp$ is not empty do
5      $(\psi, c) = tmp.pop()$
6      if $\psi$ has an $x_i$-expression then
7          $\theta =$ an $x_i$-expression of $\psi$
8          for $(\theta', c') \in EquivSplit(\theta, c, i)$ do
9              $\psi' = \psi[\theta \leftarrow \theta']$
10             $\psi' = \psi'[\delta(x_i) \leftarrow d]$
11             $tmp[\psi'] = tmp[\psi'] \cup c'$
12     else
           // $\psi$ does not depend on $x_i$
13         $res[\psi] = res[\psi] \cup c$
14 return $res$
```

The recursive call on line 9 in *EquivSplit* at step i uses parameter $i+1$; since i is bounded by $|X|+1$ (height of the decision diagram), this recursion terminates. *SolveSub* recursively solves strict sub-expressions of ϕ, hence the recursion is bounded by the height of the syntactic tree. Since calls to *EquivSplit* from *SolveSub* always concern strictly smaller expressions, the mutual recursion is also bounded.

Both algorithms work by successively refining and coarsening a partition of the input set. Any time a pair (ψ, c) is inserted into the output, ψ is obtained by evaluating ϕ (or a derivative) on elements of c. Since the output is stored in a map, merging cells (ψ, c) and (ψ', c') respects the constraint that $\psi = \psi'$, hence c and c' belong to the same equivalence class.

The full proof is presented as an appendix.

Complexity of *EquivSplit*. In Algorithm 1, the α_d's for the loop on line 6 are already provided by the DDD representation of valuations, so that this loop is a just a walk of already computed sets. The main source of complexity in this function lies in the call to *SolveSub*. In the case when ϕ has no nested δ operators, then the loop on line 8 has a single pass. The recursion on line 9 explores the subsequent part of the DDD, so that, using a cache, the total complexity of *EquivSplit* is related to the size of the input DDD, rather than to the size of V.

Complexity of *SolveSub*. The look-ahead of *SolveSub*, performed on line 8 of function 2, refines the α_d in input. This refinement (that builds new decision diagrams) can be arbitrarily fine, and depends on the input expression and the input set of valuations.

The overall complexity of *SolveSub* is thus hard to predict and depends on the number of equivalence classes built.

A worst case for our technique would be an expression computing a hash value based on the values in all the memory slots. A perfect hash function would yield equivalence classes limited to singletons, hence encountering exponential worst case complexity (linear over states contained). Conversely, expressions with a small codomain (such as boolean expressions) give a small bound on the maximum number of equivalence classes manipulated by the algorithm. A peak effect for symbolic techniques occurs when an intermediate DDD size is proportional to its set size. This may occur anytime a partition element is built, hence finer partitions are more likely to induce a peak effect.

Caches. A cache for *EquivSplit* is built by associating to each $\langle$DDD, expression$\rangle$ pair the set $\{\langle$DDD, expression value$\rangle\}$ that partitions the input DDD into equivalence classes for the input expression. The full evaluation of various statements may thus share the cache allowing computation of common sub-expressions. Because it contains partial evaluations results, and no specific attempt is made to reconcile combined results, the structure of this cache differs from a decision diagram representing the full effects of transitions, although it allows to reconstruct the same transition information.

Variable Order. Much of the complexity for both of these algorithms depends on the variable ordering used in the DDD encoding. The equivalence classes depend on the order in which x_i's are visited. The representation size of the equivalence classes also strongly depends on this order. Heuristically, orderings that minimize invocations to *SolveSub* reduce the complexity. Limiting the depth of the look-ahead mechanism also helps to build DDD that share existing suffixes.

In our experiments, we adapted the FORCE algorithm [1]. Given a directed hypergraph where weighted edges represent constraints on variables (nodes of the hypergraph), FORCE heuristically computes an ordering on variables that minimizes the total weight. Expressions induce constraints on the variables in their support. By assigning a strong weight to constraints implying invocations to *SolveSub*, and small weight to constraints enforcing locality, we obtained satisfactory results.

4.5 Evaluating assignments

We now informally present how to use our new algorithms to handle assignments of expressions to memory slots. We consider a semantic of a software system is described as sequences of assignments. An assignment is a pair of expressions (ϕ, ψ), where ϕ denotes the address of the affected memory slot and ψ the new value to assign to it. Allowing ϕ to depend on current memory state allows to model assignments such as `t[i] := 0`.

In our DDD implementation, an assignment is encoded as a homomorphism $\mathbb{D} \mapsto \mathbb{D}$. It evaluates both ϕ and ψ by walking the input DDD. As variables are encountered, ϕ and ψ are partially evaluated. If dependencies on current variable are not eliminated (nested δ), *SolveSub* is invoked. At some point, ϕ is reduced to a constant, which is the target of the assignment. When this target is reached, ψ must then be evaluated to a constant which may involve a look ahead using *SolveSub*.

Since our assignments are encoded as homomorphisms, they benefit from the automatic rewriting rules of [10]. These rules use the support of the expressions to skip don't-care variables and build clusters of transition effects. Our algorithm can also be implemented within other DD libraries. However, using DDD and homomorphisms allowed us to immediately benefit from these features.

5 Assessment

We compare our approach to related work and assess its efficiency compared to other symbolic techniques.

5.1 Related work

To encode a transition, the original symbolic approach [5] relies on a second set of variables that associates to each variable its new state after the transition, for all potential states. The global transition relation is then the monolithic union (logical or of behaviours) of all possible transitions. This monolithic approach matches the synchronous semantics of hardware systems, but yields intractable representations in many cases.

This forced to introduce new strategies [13], where an explicitly managed set of DD store conjuncts of the transition relation. This process, called transition clustering, allows to overcome some of the limits of the monolithic approach.

For Globally Asynchronous Locally Synchronous (GALS) systems, [6], proposes to design the clustering according to the top-most variable in transition supports. The semantics of such systems is given as an asynchronous interleaving of locally synchronous actions (e.g. Petri nets). Such a clustering allows *saturation* to optimize the evaluation of the least fixpoint of a set of conjuncts: based on the interleaving semantics of the conjuncts, the fixpoint is first computed on lower parts of the DD.

A similar formalism is proposed by LTSmin [4]. A system is defined as consisting of k state variables with a discrete domain D and of transitions described primarily by their support composed of $k' \leq k$ variables. To compute the state space, LTSmin relies on third-party existing explicit model-checkers that provide a computation procedure called for each encountered value of the support in the global state space. Thanks to this projection, the number of these calls is bounded by $D^{k'}$ and in practice is limited to actually encountered states. This tool also implements state-of-the-art symbolic techniques, such as saturation, using classical encoding with two "before" and "after" variables per system state variable.

This approach is however severely challenged when the support grows. If the high-level model features array manipulation, pessimistic assumptions on the supports end up with supports including most (if not all) state variables. In such an extreme case, the explicit engine is invoked at least once for each state, negating any possible gain from the use of DD. Additionally, such individual insertion of paths in a DD is liable to produce exponential memory peak effects. Large supports also severely limit the possibilities of saturation as clusters are based on the support of transitions.

The algorithms we present in this paper partly overcome these difficulties. Large supports are often the result of array manipulation or composition of local effects induced by sequences of assignments. As we have seen, the support of an expression is dynamically reduced. Large potential supports due to array manipulations are correctly resolved on-the-fly by *EquivSplit*. Compositions of effects are managed as explicit composition of homomorphisms, each of which has a support defined by its underlying expressions. Our fully symbolic encoding of the expressions avoids any explicit step where states are individually considered in the model-checking algorithm.

5.2 Implementation

To assess our algorithms, we chose to use benchmark models from the BEEM database [12], that are written in the Divine language [2]. To this end, we defined an intermediate formalism called Guarded-Action Language (GAL)[3], that can be manipulated symbolically with the algorithms described in this paper. This formalism defines a system's memory μ using integer variables and fixed size arrays of integers. Its transitions are composed of a guard that is a boolean expression over variables and a sequence of statements that are assignments of expressions to variables (or to cells of an array). A state of a GAL system is defined as the valuation of all variables. A transition is enabled in any state where the guard is true. Firing an enabled transition yields in a single step the successor state obtained by executing the assignments of the transition in an atomic sequence. The semantics are thus globally asynchronous, but sequences of statements are locally synchronous, reflecting the semantics of concurrent systems.

This small formalism offers a rich signature Σ consisting of all C operators for manipulation of the `int` data type and of arrays (including nested array expressions). There is no explicit support for pointers, though they can be simulated with an array *heap* and indexes into it. It also supports full C-like boolean expressions.

With these features, translation of Divine models into GAL was relatively straightforward. This technical work was done by adapting the code of LTSmin's wrapper for Divine models, where the semantic bridge to a system based on integer variables already existed. Divine is a language for describing processes that communicate through bounded channels, shared variables and/or synchronization. Channels are modeled using arrays. Synchronizations use a conjunction of local condition as a guard, and a sequence of local effects on each process as action. Priorities (deriving from the "commit" semantics of Divine) are enforced by adding the negation of the disjunction of the guards of higher priority to guards of transitions with lower priority.

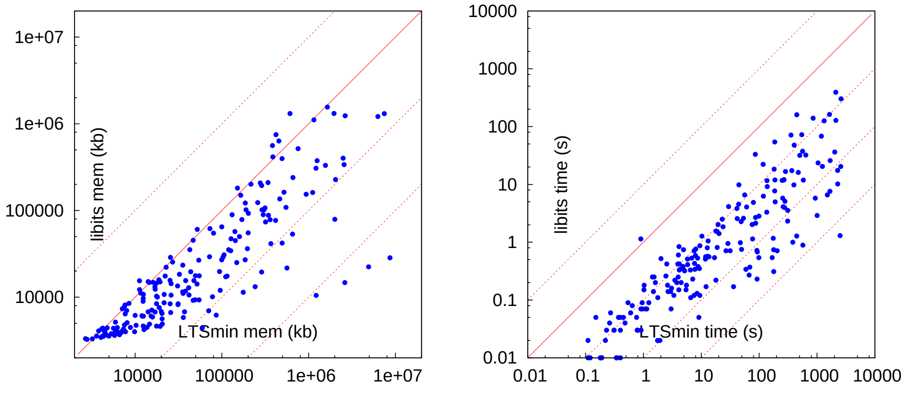
5.3 Performance Assessment

To assess our new technique we built an extension to the `libits` tool[4], and compared its performance to classical state-of-the-art approaches, represented by the tools LTSmin [4], `super_prove` [3]. The performance comparison is based on the full set of models from the BEEM database [12]. Here we only report on reachability properties that were also provided in the context of a recent hardware model checking contest (HWMCC'12[5]) as SAT instances. Our implementation supports full CTL and LTL

[3] `http://move.lip6.fr/software/DDD/gal.php`

[4] `http://ddd.lip6.fr`

[5] `http://fmv.jku.at/hwmcc12`



(a) comparison on peak memory (b) comparison on time

Fig. 2. `libits` vs LTSmin, same variable ordering

model-checking of Divine models. All experiments were run on a Xeon 64 bits at 2.6 GHz processor.

LTSmin is a tool suite for model-checking, that implements state-of-the-art symbolic techniques (see 5.1). It can use several third-party DD libraries, but we configured it to use DDD to allow easier algorithmic comparison. Indeed the state encoding being provided by the same DDD library as ours, the main difference between this tool and ours is the use of *EquivSplit*.

`super_prove` is a SAT based model-checker. It was the winner of the "single safety/ bad-state property" track of the HWMCC'12, that contains the BEEM models. It is thus, to our knowledge, the best SAT-solver for this particular benchmark. SAT techniques are very different from those discussed in this paper, but raw performance comparisons on this benchmark are still possible.

`libits` is a DD-based verification library that uses both hierarchical set decision diagrams and DDD to support model-checking (CTL, LTL) of composition of labeled transition systems described symbolically. Transition systems can be described using several input formalisms, such as labeled discrete time Petri nets. The GAL formalism was embedded in this framework but only uses DDD.

Detailed results of experiments are presented as scatter plots comparing two tools over the whole benchmark. Each point represents a (model,formula) pair that was tested for reachability with both tools. A point below the diagonal means that `libits` is more efficient than the other tool. Our plots use a logarithmic scale. Lines parallel to the diagonal represent performance ratios of 10, 100 . . . (resp. 0.1, 0.01 . . .).

models tested	treated by libits	treated by LTSmin	treated by both	treated by none
293	264	212	197	22

Table 1. `libits` vs LTSmin

`libits` vs. LTSmin. We compare the performance for the generation of the state space of the models, with 1 hour and 10Gb containment. Statistics of the results are shown in Fig. 1, and more detailed results are shown on Fig. 2. The results confirm that our *EquivSplit* algorithm performs better than the classical symbolic approach. With the same implementation of DDD and the same variable ordering, our implementation is up to 1000 times faster and 100 times less memory consuming than LTSmin. For a dozen models, LTSmin is slightly more memory efficient than libits, but this can be attributed to side-effects of the garbage collection policy.

`libits` vs. `super_prove`. We compare the performance of both `libits` and `super_prove`, with a containment of 1Gb in memory and 900 seconds wall clock time (`super_prove` uses 4 cores while `libits` is mono-threaded). These are the containment settings used in the HWMCC competition. Summary of the results are shown in Fig 4, and detailed results are presented on Fig. 3. We only compared the time usage, since the memory consumption for SAT techniques is usually insignificant. Note that all `libits`'s fails are due to a memory overflow, whereas all `super_prove`'s fails are due to a time overflow.

`libits` treats about 35% more models than `super_prove`. Also, `libits` is quicker than `super_prove` for 80% of the models treated by both tools, with a speed-up factor up to 1000. On the other models, `super_prove`'s speed-up factor ranges up to 100.

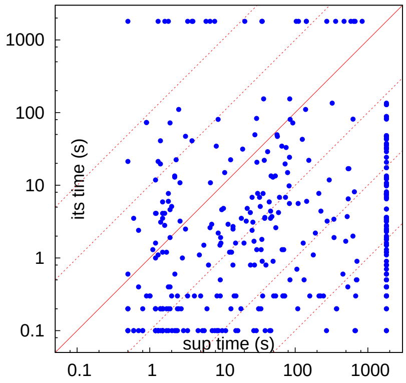


Fig. 3. Time comparison between `libits` and `super_prove`

	# unsat # unsat	mean time unsat (s)	# sat # sat	mean time sat (s)
`libits`	184	14.6	192	8.6
`super_prove`	112	140.6	170	45.1

models tested	treated by `libits`	treated by `super_prove`	treated by both	treated by none
456	376	282	258	56

Fig. 4. `libits` vs `super_prove` (top: mean runtime and bottom: number of instances solved)

The top table in Fig. 4 shows that `libits` runs on average 5 times faster on satisfied properties and 10 times faster on unsatisfied properties than `super_prove` that stops as soon as it finds a solution for satisfied instances. Our tool regularly interrupts the computation to check whether a solution exists in the states computed so far. When these checks are deactivated, `libits` is 4 times faster on satisfied properties and 14 times faster on unsatisfied properties. Unsatisfied instances require both tools to explore the whole reachability graph: these are the hardest problems.

On this benchmark, we show that state-of-the-art symbolic manipulation of decision diagrams can still outperform the best SAT-based techniques.

6 Conclusion

This paper proposes a new algorithm, *EquivSplit*, that allows more efficient symbolic manipulation of software-like models. It uses equivalence relations to avoid explicit manipulation of states. Assessment on a large third-party benchmark shows that this approach improves existing decision diagram-based techniques, and can outperform SAT-based ones.

Our algorithm supports arbitrary signatures (languages), and can be used with any type of decision diagrams. It uses information provided by the high-level expressions of the transition relation to dynamically optimize computations.

Using the *EquivSplit* algorithm, we are currently investigating the combination of symmetries with decision diagrams as an extension of previous work performed without this contribution [8].

References

1. Aloul, F., Markov, I., Sakallah, K.: Force: a fast and easy-to-implement variable-ordering heuristic. In: 13th ACM Great Lakes symposium on VLSI. pp. 116–119. ACM (2003)
2. Barnat, J., Brim, L., Češka, M., Ročkai, P.: DiVinE: Parallel Distributed Model Checker (Tool paper). In: Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC). pp. 4–7. IEEE (2010)
3. Berkeley Logic Synthesis and Verification Group: ABC: A System for Sequential Synthesis and Verification, Release 12/10/06, `http://www.eecs.berkeley.edu/~alanmi/abc/`
4. Blom, S., van de Pol, J., Weber, M.: Ltsmin: Distributed and symbolic reachability. In: Computer Aided Verification. pp. 354–359. Springer (2010)
5. Burch, J., Clarke, E., et al.: Symbolic model checking: 10^{20} States and beyond. Information and computation 98(2), 142–170 (1992)
6. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. Tools and Algorithms for the Construction and Analysis of Systems pp. 379–393 (2003)
7. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
8. Colange, M., Kordon, F., Thierry-Mieg, Y., Baarir, S.: State Space Analysis using Symmetries on Decision Diagrams. In: 12th International Conference on Application of Concurrency to System Design (ACSD). pp. 164–172. IEEE Computer Society (Jun 2012)
9. Couvreur, J., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.: Data decision diagrams for petri net analysis. Application and Theory of Petri Nets pp. 129–158 (2002)
10. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Hierarchical set decision diagrams and automatic saturation. Applications and Theory of Petri Nets pp. 211–230 (2008)
11. Holzmann, G.J.: The model checker spin. IEEE Transactions on Software Engineering 23, 279–295 (1997)
12. Pelánek, R.: Beem: Benchmarks for explicit model checkers. In: Model Checking Software, 14th Int'l SPIN Workshop. LNCS, vol. 4595, pp. 263–267. Springer (2007)
13. Ranjan, R., Aziz, A., Brayton, R., Plessier, B., Pixley, C.: Efficient bdd algorithms for fsm synthesis and verification. IWLS95, Lake Tahoe, CA 253, 254 (1995)

A Proofs of correctness

To prove the correctness of our algorithms, we need some additional definitions and notations.

Definitions and notations. For $1 \leq i \leq |X|+1$, $Y_i = \{x_i, \ldots, x_{|X|}\}$.

We define an order over the expressions for induction on expressions in the proof.

Definition 6. *Let ϕ and ψ be two expressions. $\phi < \psi$ if and only if ϕ has less symbols than ψ.*

Definition 7. *Let $x \in X$. We define the equivalence relation $\wr_x$ over valuations as:*

$$\mu \wr_x \mu' \Leftrightarrow \mu(x) = \mu'(x)$$

Definition 8. *Let μ be a valuation, x be an address, ϕ and ϕ' two expressions.*
We define $\phi \rightsquigarrow_{x,\mu} \phi'$ if and only if one of the following conditions holds:

- *$\phi' = \phi[\psi \leftarrow eval(\mu, \psi)]$, where ψ is an x-expression of ϕ whose support is not empty;*
- *ϕ contains at least one constant x-expression and $\phi' = eval(\mu_{|\{x\}}, \phi)$.*

$\rightsquigarrow^*_{x,\mu}$ denotes the reflexive and transitive closure of $\rightsquigarrow_{x,\mu}$. It is easy to see that if $\phi \rightsquigarrow_{x,\mu} \phi'$, then the number of occurrences of δ in ϕ' is strictly less than in ϕ. Therefore, there is no infinite sequence $\phi \rightsquigarrow_{x,\mu} \phi_1 \rightsquigarrow_{x,\mu} \ldots$, and thus, for any expression ϕ, there exists an expression ϕ' that does not depend on x such that $\phi \rightsquigarrow^*_{x,\mu} \phi'$.
We now choose a deterministic criterion for choosing the x-expression ψ in the first case of definition 8. For example, ψ is the first x-expression encountered during a depth-first walk of the syntactic tree of ϕ. This ensures the uniqueness of the above ϕ': for any expression ϕ, there exists a unique expression, denoted by $norm(\mu, x, \phi)$ that does not depend on x and such that $\phi \rightsquigarrow^*_{x,\mu} norm(\mu, x, \phi)$.
$norm(\mu, x, \phi)$ is computable by repeatedly evaluating the x-expressions of ϕ chosen according to this criterion.

Definition 9. *Given an expression ϕ, we define the equivalence relation $\equiv^x_\phi$ over valuations as follows:*

$$\mu \equiv^x_\phi \mu' \Leftrightarrow norm(\phi, x, \mu) = norm(\phi, x, \mu')$$

Note that the choice of the criterion to pick an x-expression of ϕ influences the structure of the equivalence classes of $\equiv^x_\phi$. This does not affect correctness of the algorithms, but may impact their complexities, as already mentioned in 4.3.

Lemma 3. *Let ϕ be an expression, μ be a valuation and x be an address.*

$$eval(\mu, norm(\mu, x, \phi)) = eval(\mu, \phi)$$

Proof. Let us consider the relation $\rightsquigarrow_\mu = \cup_{y \in X} \rightsquigarrow_{y,\mu}$. If $\phi \rightsquigarrow_\mu \phi'$, then the number of occurrences of δ in ϕ' is strictly less than in ϕ. Therefore, there is no infinite sequence $\phi \rightsquigarrow_\mu \phi_1 \rightsquigarrow_\mu \ldots$, and there exists an expression ϕ' that depends on no variable and such that $\phi \rightsquigarrow^*_\mu \phi'$. It is easy to see that $\phi' = eval(\mu, \phi)$. Therefore, since $eval(\mu, \phi)$ is well-defined, ϕ' is unique, and does not depend on the path leading from ϕ to ϕ'.

Similarly, $\phi'' = eval(\mu, norm(\mu, x, \phi))$ depends on no variable and $norm(\mu, x, \phi) \rightsquigarrow^*_\mu \phi''$. Since $\rightsquigarrow_{x,\mu} \subseteq \rightsquigarrow_\mu$, we have $\phi \rightsquigarrow^*_\mu norm(\mu, x, \phi) \rightsquigarrow^*_\mu \phi''$. By unicity of ϕ', we conclude that $\phi' = \phi''$, hence $eval(\mu, norm(\mu, x, \phi)) = eval(\mu, \phi)$.

Correctness of the algorithms. We prove the correctness of both *EquivSplit* and *SolveSub*, using a double induction on parameters i and ϕ.

$\mathit{EquivSplit}(\phi,V,i)$ returns a set of couples $\{(\phi_1,c_1),\ldots,(\phi_n,c_n)\}$ such that $c_1,\ldots,c_n$ are the equivalence classes of $\sim_\phi^{Y_i}$ on V, and $\phi_j = \mathit{eval}(\mu,\phi)$ for any $\mu \in c_j$, provided that ϕ does not depend on $x_1,\ldots,x_{i-1}$.

$\mathit{SolveSub}(\phi,V,i)$ returns a set of couples $\{(\phi_1,c_1),\ldots,(\phi_n,c_n)\}$ such that $c_1,\ldots,c_n$ are the equivalence classes of $\equiv_\phi^{x_i}$ on V, and $\phi_j = \mathit{norm}(\mu,x_i,\phi)$ for any $\mu \in c_j$, provided that ϕ does not depend on $x_1,\ldots,x_{i-1}$. Note that all elements in V must agree on the value of x_i: V must be contained in an equivalence class of $\wr_{x_i}$.

Induction on i. If $i = |X|+1$, then any expression ϕ that does not depend on $x_1,\ldots,x_{|X|}$ is constant. Then both $\sim_\phi^{Y_i}$ and $\equiv_\phi^{x_i}$ are trivial. Furthermore, $\mathit{eval}(\mu,\phi) = \phi$ and $\mathit{norm}(\mu,x_i,\phi) = \phi$ for all valuations μ. $\mathit{EquivSplit}(\phi,V,i)$ and $\mathit{SolveSub}(\phi,V,i)$ both return the singleton $\{(\phi,V)\}$, ensuring their correctness.

Let $1 \leq i \leq |X|$. Let us assume that both $\mathit{EquivSplit}(\psi,V,i+1)$ and $\mathit{SolveSub}(\psi,V,i+1)$ are correct for any set V and expression ψ. We prove, by structural induction on ϕ, that $\mathit{EquivSplit}(\phi,V,i)$ and $\mathit{SolveSub}(\phi,V,i)$ are correct.

Induction on ϕ. Let ϕ be an expression. If ϕ is a single symbol, it is a constant expression, and the conclusion is the same as above for the base case of the induction on i.

Let us now suppose that ϕ is not reduced to a single symbol, and that (induction hypothesis) $\mathit{EquivSplit}(\psi,V,i)$ is correct for all V and $\psi < \phi$, and $\mathit{SolveSub}(\psi,V,i)$ is correct for all V and $\psi < \phi$. We first prove that $\mathit{SolveSub}(\phi,V,i)$ is correct, then that $\mathit{EquivSplit}(\phi,V,i)$ is correct.

SolveSub is based on a while loop, that has three invariants, that we prove:

$$\forall(\psi,c) \in \mathit{tmp}, \forall\mu \in c, \phi \rightsquigarrow^*_{x_i,\mu} \psi \tag{1}$$

$$\forall(\psi,c) \in \mathit{res}, \forall\mu \in c, \psi = \mathit{norm}(\mu,x_i,\phi) \tag{2}$$

$$\text{all the sets stored in } \mathit{tmp} \text{ and } \mathit{res} \text{ form a partition of } V \tag{3}$$

These conditions are obviously true when entering the loop for the first time (initializations of lines $1-3$).

When entering the loop, a couple (ψ,c) is first removed from tmp (line 5). By (1), $\phi \rightsquigarrow^*_{x_i,\mu} \psi$ for all $\mu \in c$.

If ψ has no x-expression, then ψ is in normal form and $\psi = \mathit{norm}(\mu,x_i,\phi)$ for all $\mu \in c$. Thus, adding (ψ,c) to *res* preserves all three invariants (lines $12-13$).

Otherwise, the algorithm picks an x_i-expression θ in ψ (line 7). θ is a proper subexpression of ψ, and, since ψ does not depend on $x_1,\ldots,x_{i-1}$, neither does θ. Thus, by induction hypothesis, $\mathit{EquivSplit}(\theta,c,i)$ is correct. Therefore, on line 8, c' is an equivalence class for $\sim_\theta^{Y_i}$ on c and $\theta' = \mathit{eval}(\mu_{|Y_i},\theta)$ for any $\mu \in c'$. Thus, $\psi' = \psi[\theta \leftarrow \mathit{eval}(\mu_{|Y_i},\theta)]$ for $\mu \in c'$ is well-defined on c' (line 9). Furthermore, since all valuations in V agree on x_i, then $\psi' = \mathit{eval}(\mu_{|\{x_i\}},\psi[\theta \leftarrow \mathit{eval}(\mu_{|Y_i},\theta)])$ is also well-defined on c' (line 10). Furthermore, $\psi \rightsquigarrow_{x_i,\mu} \psi'$ or $\psi \rightsquigarrow^2_{x_i,\mu} \psi'$, and thus $\phi \rightsquigarrow^*_{x_i,\mu} \psi'$ for all $\mu \in c'$. Since the equivalence classes of $\sim_\theta^X$ partition c, adding such pairs (ψ',c') to tmp preserves the

three invariants.
Since $\psi \rightsquigarrow^{+}_{x_i,\mu} \psi'$, ψ' is strictly closer to any of its normal forms than ψ. This ensures termination of the loop.
Finally, the loop ends when tmp is empty. Thus, by (3), the sets stores in *res* partition V, and using (2), we conclude that *res* contains exactly the equivalence classes of $\equiv^{x_i}_{\phi}$ on V.

We now prove that $\mathit{EquivSplit}(\phi, v, i)$ is correct. It is based on three nested loops. The innermost loop has two invariants, that we prove below:

$$\forall(\psi, c) \in res, \forall\mu \in c, \phi = eval(\mu_{|Y_i}, \psi) \tag{4}$$

$$\forall(\psi_1, c_1), (\psi_2, c_2) \in res, \phi_1 = \phi_2 \Rightarrow c_1 = c_2 \tag{5}$$

These invariants are obviously true before entering the loop on line 6. The α_d's defined on line 5 are the equivalence classes of $\wr_{x_i}$ on V. Thus $\theta = eval(\mu_{|x_i}, \phi)$ (line 7) is well-defined on α_d. Furthermore, $\phi \rightsquigarrow^{*}_{x_i,\mu} \theta$ for any $\mu \in \alpha_d$. θ is either an expression smaller than ϕ, or ϕ. In both cases, by induction hypothesis or by the proof of correctness of $\mathit{SolveSub}(\phi, V, i)$ above, $\mathit{SolveSub}(\theta, V, i)$ is correct.

On line 8, c is an equivalence class for $\equiv^{x_i}_{\theta}$ on α_d and $\psi = norm(\mu, x_i, \theta) = norm(\mu, x_i, \phi)$ (since $\phi \rightsquigarrow^{*}_{x_i,\mu} \theta$) for any $\mu \in c$. Thus ψ does not depend on x_i, and, by induction hypothesis, $\mathit{EquivSplit}(\psi, c, i+1)$ is correct. Therefore, on line 9, c' is an equivalence class for $\sim^{Y_{i+1}}_{\psi}$ on c, and $\psi' = eval(\mu_{|Y_{i+1}}, \psi)$ for any $\mu \in c'$. Let $\mu \in c'$.

$$\begin{aligned}
\psi' &= eval(\mu_{|Y_{i+1}}, \psi) \\
&= eval(\mu_{|Y_{i+1}}, norm(\mu, x_i, \phi)) \\
&= eval(\mu_{|Y_i}, norm(\mu, x_i, \phi)) && \text{because } norm(\mu, x_i, \phi) \text{ does not depend on } x_i \\
&= eval(\mu_{|Y_i}, \phi) && \text{by lemma 3}
\end{aligned}$$

When inserted in *res* (line 10), c' is merged to a $(\psi'', c'') \in res$ if, and only if, $\psi'' = \psi'$. By invariant 4, $\psi'' = eval(\mu_{|Y_i}, \phi)$ for $\mu \in c''$, so that this insertion in *res* preserves both invariants.

Recall that α_d (line 6) is an equivalence class for wr_{x_i}. c being an equivalence class for $\equiv^{x_i}_{\theta}$ on α_d, it is an equivalence class for $\wr_{x_i} \cap \equiv^{x_i}_{\theta}$ on V. Similarly, c' is an equivalence class for $\sim^{Y_{i+1}}_{\psi}$ on c, hence an equivalence class of $\wr_{x_i} \cap \equiv^{x_i}_{\theta} \cap \sim^{Y_{i+1}}_{\psi}$ on V. The algorithm reaches line 11 when all these equivalence classes of $\wr_{x_i} \cap \equiv^{x_i}_{\theta} \cap \sim^{Y_{i+1}}_{\psi}$ on V have been treated and inserted into *res*. Thus, the union of the sets stored in *res* is exactly V. By invariant 4, they are pairwise disjoint, hence form a partition of V. By invariants 4 and 5, we conclude that they are exactly the equivalence classes of $\sim^{Y_i}_{\phi}$, proving the correct result of $EquivSplit(\phi, V, i)$.