Representing and Approximating Transfer Functions in Abstract Interpretation of Hetereogeneous Datatypes

B. Jeannet

INRIA–IRISA

Abstract. We present a general method to combine different datatypes in Abstract Interpretation, within the framework of verification of reactive system. We focus more precisely on the efficient representation and approximation of the transfer functions involved in the abstract fixpoint computations. The solution we propose allows to tune smoothly the necessary tradeoff between accuracy and efficiency in the analysis.

1 Introduction

We are interested in the verification by Abstract Interpretation [8] of reactive systems with infinite state space, generated by a set of variables of different types (enumerated types, integers, FIFO queues, ...). Combining different datatypes in Abstract Interpretation is however difficult, especially in the case where a *relational* abstract domain is needed and when some of the involved datatypes have an infinite domain. For instance, how to represent efficiently the logical relation $b \Rightarrow (x < 0) \vee (x > 10)$, where b is a Boolean and x a real ? And, during a fixpoint computation, how should we compute its image by the vector of transition functions $\begin{pmatrix} b \\ x \end{pmatrix} \longmapsto \begin{pmatrix} b \vee (x>1) \\ \textbf{if } b \wedge (x>5) \textbf{ then } 2x \textbf{ else } x-1 \end{pmatrix}$? Efficiency is here a major issue: for instance synchronous programs [12] that have dozens of Boolean and numerical variables are still small programs. Their transition functions can nevertheless involve many conditions.

[17] proposed a solution based on the partitioning of a basic non-relational abstract domain to represent relationship between datatypes. However it did not focus on the efficient representation and computation of the symbolic transition functions of the analysed system, which is the object of this paper. For this purpose [16] describes a technique inspired by compilation techniques of the Lustre synchronous language [13], and applies it to the verification of Lustre programs with both Boolean and numerical variables.

In this paper we generalize this method, formalize it in an abstract interpretation framework and analyse complexity and accuracy issues. This generalization allows to combine any datatypes, provided that suitable abstract domains are available for each of them. This method has lead to considerable performance improvement without being penalized by the loss of precision. For the application we have in mind, which is the analysis of reactive systems, an important

M. Hermenegildo and G. Puebla (Eds.): SAS 2002, LNCS 2477, pp. 52–68, 2002.

point is that the *performance problem* is often as important as the *precision* problem.

Related Work. Several generic operators for composing abstract domains have been proposed in the litterature. The *direct* and *reduced product* [9] lead to domains which are non-relational, and where the image $f(x)$ of a concrete element x by the function f is abstracted by the pair $(f^{\alpha_1}(x^{\alpha_1}), f^{\alpha_2}(x^{\alpha_2}))$: the components do not interact during the computation, which leads in our case to a very coarse approximation. In order to solve this latter limitation, the *open product* [7] uses a notion of *query* which allows f^{α_1} for instance to check if a property on x^{α_2} is satisfied. However, it does not allow to split x^{α_2} w.r.t. to the satisfaction of the property. It should also be noted that these products were mainly intended for combining two abstract domains of the same concrete domain, rather than unrelated domains like $\mathbb{B}^n$ and $\mathsf{Pol}(\mathbb{R}^p)$. Only the *tensor product* [21,22] leads to a truly relational domain. It exhibits the nice property that $\wp(D_1) \otimes \wp(D_2) = \wp(D_1 \times D_2)$. If we use it to combine the Boolean lattice $\mathbb{B}^n$ with the lattice of convex polyhedra $\mathsf{Pol}(\mathbb{R}^p)$, we get $\mathbb{B}^n \otimes \mathsf{Pol}(\mathbb{R}^p) = \mathbb{B}^n \to \mathsf{Pol}(\mathbb{R}^p)$, which is actually the lattice used in [14]. We however argued in [17] that its complexity in $\mathcal{O}(2^n)$ limits too strongly the number n of Boolean variables taken into account in the analysis. Moreover it is not clear that the tensor product of infinite lattices is computable (consider $\mathsf{Pol}(\mathbb{R}^p) \otimes \mathsf{Reg}(\Sigma)$, where $\mathsf{Reg}(\Sigma)$ is the lattice of regular languages over Σ).

In the verification community, combining enumerated types with an infinite type (counters, clocks, FIFO queues (e.g. [1,15,4]) is common. In many cases however, finite-state variables are entirely put in the control structure and there is only one infinite type: the verification algorithms manipulate then sets of values of the same type. Among the works aimed at combining different types, [18, 19] use BDDs where some variables are associated to numerical constraints to combine Boolean and numerical datatypes. [5,25] uses instead formula in a kind of disjunctive form. These proposals can be viewed as instances of the *disjunctive completion* technique [10] applied to the direct product of the domains associated to each datatype. The common philosophy of these works is to perform *exact* computations as much as possible and to accept approximations mainly in acceleration and/or widening operators. We give a more precise comparison of these works with ours in the section 5.

In the constraint solving community, there exists classical methods to combine decision procedures, for instance [20,23]. However they are usually restricted to quantifier-free formulas, which prevents the computation of postconditions and allows the computation of preconditions only in the case of closed systems (by using substitution). Their use in verification would also require a suitable widening operator.

Outline of the paper. We first present in section 2 the model of the programs we want to verify, and in section 3 our analysis framework. Section 4 discusses briefly how to manipulate transition functions. Section 5 presents the basic methods to compute transfer functions. Section 6 presents our method based on a combina-

tion of the basic methods. Section 7 explains how to perform partial evaluation and its usefulness on a partitioned system. We present in section 8 some experiments before concluding.

2 The Model of the Considered Programs

We consider in this paper programs that have the following structure: they have only a finite set of control points (defined by a "program counter" variable) and a finite set of variables of different types (Boolean and enumerated types, numbers, queues over some alphabet, ...).

Definition 1 (Types, domains & predicates). *We assume a set* $\mathcal{T} = \{t_1, \ldots, t_{|\mathcal{T}|}\}$ *of* variable types*; to each type* $t \in \mathcal{T}$ *is associated:*

- *a domain* D_t*, in which variables of type* t *take their values;*
- *for any* $k \geq 0$*, a set* $\mathcal{C}_t^{(k)}$ *of* elementary predicates $X \subseteq D_t^k$*;*
- *for any* $k \geq 0$*, a set* $\mathcal{E}_t^{(k)}$ *of* elementary functions $D_t^k \to D_t$*;*

The intuition behind this definition is that elementary predicates and functions are easy to abstract and will be used as building blocks to define more complex predicates and functions.

Example 1. In the case of synchronous programs considered in [17,16], we had $\mathcal{T} = \{\text{bool}, \text{int}\}$ and

$$\left\{ \begin{array}{ccc} D_{\text{bool}} = \mathbb{B}, & \mathcal{C}_{\text{bool}}^{(k)} = \{b_i, \bar{b_i}\}_{1 \leq i \leq n}, & \mathcal{E}_{\text{bool}}^{(k)} = \mathbb{B}^k \to \mathbb{B} \\ D_{\text{int}} = \mathbb{Q}, & \mathcal{C}_{\text{int}}^{(k)} = \mathsf{LinCons}(\mathbb{Q}^k), & \mathcal{E}_{\text{int}}^{(k)} = \mathsf{LinExpr}(\mathbb{Q}^k) \end{array} \right\}$$

where $\mathsf{LinExpr}(\mathbb{Q}^k)$ is the set of affine expressions and $\mathsf{LinCons}(\mathbb{Q}^K)$ the set of linear constraints on $\mathbb{Q}^K$. In other words, all Boolean predicates are allowed, whereas elementary numerical predicates are linear constraints and elementary numerical functions are affine functions. □

For any vector $\boldsymbol{n} = (n_1, \ldots, n_{|\mathcal{T}|})$ of natural numbers and for any $t_k \in \mathcal{T}$, n_{t_k} denotes n_k. We note $D^{\boldsymbol{n}}$ the set $\prod_{t \in \mathcal{T}} D_t^{n_t}$, which will be the state space of the program. For any element $d \in D^{\boldsymbol{n}}$, d_t denotes its projection on the domain $D_t^{n_t}$. We will consider in the sequel the following kind of *conditional functions*, built by combining elementary predicates and functions.

Definition 2 (Conditional functions). *The set* $\mathcal{F}_t^{(\boldsymbol{n})}$ *of* conditional functions *of type* $D^{\boldsymbol{n}} \to D_t$ *is defined by the syntax:*

$$\begin{array}{l} \text{fun}_t ::= \mathit{expr}_t \mid \textbf{if } \mathit{cond} \textbf{ then } \text{fun}_t \textbf{ else } \text{fun}_t \\ \mathit{expr}_t \in \mathcal{E}_t^{(n_t)} \quad , \quad \mathit{cond} \in \bigcup_{t \in \mathcal{T}} \mathcal{C}_t^{(n_t)} \end{array}$$

Intuitively, conditional functions are elementary functions guarded by predicates on variables of any type. The value of a function of a certain type may depend on

values of other types only through tests on elementary predicates. The semantics of $f \in \mathcal{F}_t^{(\boldsymbol{n})}$ is the obvious one:

$$[\![\textbf{if } c \textbf{ then } f^+ \textbf{ else } f^-]\!] = \lambda d \in D^{\boldsymbol{n}} \,.\, \mathit{Ite}([\![c]\!](d), [\![f^+]\!](d), [\![f^-]\!](d))$$
$$[\![\mathit{expr}_t \in \mathcal{E}_t^{(n_t)}]\!] = \lambda d \,.\, \mathit{expr}(d_t) \quad , \quad [\![\mathit{cond} \in \mathcal{C}_t^{(n_t)}]\!] = \lambda d \,.\, (d_t \in \mathit{cond})$$

Example 2. In the previous example, if b_1 and b_2 are Boolean variables and x_1, x_2 are integer variables, then $f(\boldsymbol{b}, \boldsymbol{x}) = (\textbf{if } b_1 \wedge b_2 \textbf{ then } x_1 + x_2 \textbf{ else } x_1)$ is a valid conditional function, but $g(\boldsymbol{b}, \boldsymbol{x}) = (\textbf{if } b_1 \wedge (x_2 \geq 0) \textbf{ then } x_1 + x_2 \textbf{ else } x_1)$ is not valid; it can however be rewritten $g(\boldsymbol{b}, \boldsymbol{x}) = (\textbf{if } b_1 \textbf{ then } (\textbf{if } (x_2 \geq 0) \textbf{ then } x_1 + x_2 \textbf{ else } x_1) \textbf{ else } x_1)$. The figure 1 shows other examples of conditional functions. □

We define now the structure of the programs we analyse.

Definition 3 (symbolic dynamic system). *A symbolic discrete dynamic system is a nuplet* $(V^s, V^i, \mathit{Init}, \mathit{Ast}, (f^v)_{v \in V}, \mathit{Final})$ *where:*

- $V^s = \bigcup_{t \in \mathcal{T}} V_t^s$ *is the set of* state variable*, where* V_t^s *denotes the set of state variables of type* t*; we note* $n_t^s = |V_t^s|$ *the number of state variables of type* t*, and* $\boldsymbol{n}^s$ *the corresponding vector;* V^i *is the set of input variables, for which we use similar notations;*
- $\mathit{Init} \in \mathcal{F}_{\text{bool}}^{(\boldsymbol{n}^s)}$ *is a Boolean function describing the set of* initial states*;*
- $\mathit{Ast} \in \mathcal{F}_{\text{bool}}^{(\boldsymbol{n}^s + \boldsymbol{n}^i)}$ *is a Boolean function giving in each state the set of possible inputs; it is called the* assertion *of the program;*
- *For every* $t \in \mathcal{T}$ *and variable* $v \in V_t^s$, $f^v \in \mathcal{F}_t^{(\boldsymbol{n}^s + \boldsymbol{n}^i)}$ *is a* transition function *giving the value of the variable* v *in the next state in function of the current value of state and input variables;*
- *for verification purpose,* $\mathit{Final} \in \mathcal{F}_{\text{bool}}^{(\boldsymbol{n}^s)}$ *is a* Boolean function *specifying the set of "bad" states that represents the violation of an* invariance *property.*

Such a symbolic discrete dynamic system describes a ordinay discrete dynamic system $(S, I, \mathit{Init}, \mathit{Ast}, \tau, \mathit{Final})$ where:

- $S = D^{\boldsymbol{n}^s}$ (resp. $I = D^{\boldsymbol{n}^i}$) is the state (resp. input) space;
- $\mathit{Init} = \{\boldsymbol{s} \in S \mid \mathit{Init}(\boldsymbol{s})\}$ is the set of initial states;
- $\mathit{Ast} = \{(\boldsymbol{s}, \boldsymbol{i}) \in S \times D^{\boldsymbol{n}^i} \mid \mathit{Ast}(\boldsymbol{s}, \boldsymbol{i})\}$ is the assertion;
- $\tau \subseteq S \times S$ is defined by $\tau(\boldsymbol{s}, \boldsymbol{s'}) \iff \exists \boldsymbol{i} : \tau(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'})$ where $\tau(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'}) \iff \mathit{Ast}(\boldsymbol{s}, \boldsymbol{i}) \wedge (\boldsymbol{s'} = f(\boldsymbol{s}, \boldsymbol{i})) \iff \mathit{Ast}(\boldsymbol{s}, \boldsymbol{i}) \wedge \bigwedge_{v \in V} s'_v = f^v(\boldsymbol{s}, \boldsymbol{i})$;
- $\mathit{Final} = \{\boldsymbol{s} \in S \mid \mathit{Final}(\boldsymbol{s})\}$ is the set of final states.

Input variables and the assertion formula allow to model the environment of an open system. They also allows to introduce non-determinism in an otherwise deterministic systems. Notice that the formalism presented above corresponds to the low-level semantics of dataflow synchronous languages. Most interpreted automaton formalisms can enter in this framework (for instance the models used in [4,1,6]). Imperative programs could also be described in this way, by using an explicit program counter variable and an explicit call stack variable, although it is probably not the most adequate representation.

3 Analysis Framework and Abstract Interpretation

The analysis we want to perform are computations of reachable and/or coreachable states (i.e., states leading to some *final* condition). The applications we have in mind are the computation of invariants and the verification of safety properties. We are also interested in *semantic slicing*, that is, in computing the set of states that belongs to the executions starting from an initial state and leading to a final state. We remind in this section the framework of [16].

Analysis purpose. We consider a symbolic dynamic system $(V^s, V^i, \mathit{Init}, \mathit{Ast}, (f^v)_{v\in V}, \mathit{Final})$ that generates a discrete dynamic system $(S, I, \mathit{Init}, \mathit{Ast}, \tau, \mathit{Final})$. For $X, Y \subseteq S$, let us note $\mathrm{post} = \lambda X.\{\boldsymbol{s'} \mid \exists \boldsymbol{s} \in X : \tau(\boldsymbol{s}, \boldsymbol{s'})\}$ and $\mathrm{pre} = \lambda Y.\{\boldsymbol{s} \mid \exists \boldsymbol{s'} \in Y : \tau(\boldsymbol{s}, \boldsymbol{s'}\}$ the *postcondition* and *precondition* functions. The set of *reachable* (resp. *coreachable*) states of a system is the set $\mathit{reach} = \mathsf{lfp}(\lambda X.\mathit{Init} \cup \mathrm{post}(X))$ (resp. $\mathit{coreach} = \mathsf{lfp}(\lambda X.\mathit{Final} \cup \mathrm{pre}(X))$), where lfp is the least fixpoint operator. If we want to verify a safety property, *Init* will be the set of initial states of the system, *Final* the set of states violating the property, and we want to check that $\mathit{reach} \cap \mathit{Final} = \emptyset$ or $\mathit{Init} \cap \mathit{coreach} = \emptyset$. If we want to perform slicing, *Init* is the set of starting states of the executions of interest, *Final* the set of goal states, and we want to compute $\mathit{reach} \cap \mathit{coreach}$.

The abstract domain. As the above-mentioned fixpoints, defined on the powerset of states $\wp(S)$, are generally not computable, we use the abstract interpretation theory to approximate them. We suppose that for each type $t \in \mathcal{T}$ and integer $n > 0$ we have a Galois connection $(\wp(D_t^n), \alpha_t, \gamma_t, A_t^{(n)})$. Our requirements for a domain $A(\sqsubseteq, \sqcup, \sqcap, \bot, \top)$ abstracting $\wp(S)$ where $S = D^{\boldsymbol{n}} = \prod_{t\in\mathcal{T}} D_t^{n_t}$ are the following: (i) we want to be compositional: A should be defined from the $A_t^{(n_t)}$'s; (ii) we wish a suitable tradeoff between accuracy and complexity of the abstract operations. We explained in the introduction why the "standard" product operators proposed in the Abstract Interpretation community do not satisfy the second requirement. [17] suggests as a solution to take the direct product $A^{(\boldsymbol{n}^s)} = \prod_{t\in\mathcal{T}} A_t^{(n_t^s)}$ to abstract subsets of S, and to use partitioning on this basic *non-relational* domain to establish relations between different datatypes.

Computation of transfer functions. We need suitable upper-approximations of post and pre operators to solve the fixpoint equations in the abstract domain. The best upper-approximation of these operators is often not computable or is too expensive to compute. In addition an accurate result may be worthless because most of the information it contains will be lost when merging the results with the $\sqcup$ operator on the (partitioned) domain.

The next sections present the main contribution of this paper, which is a framework to represent and compute correct approximations of the transfer functions post and pre.

4 Manipulation of Conditional Functions

We explain briefly here how we represent and manipulate conditional functions $f \in \mathcal{F}_t^{(\boldsymbol{n})}$. These functions are proper binary decision diagrams, where decisions

are taken depending on the truth value of elementary predicates, and where their results are elements of type D_t. So the choice of MTBDDs (Multi-Terminal Binary Decision Diagrams) [2] which is an efficient representation of decision diagrams, is very natural. The use of BDDs and/or MTBDDs, the nodes of which are labelled by predicates (or constraints), has been studied in several papers [18,19,3]. Our own use differs however in one point: we use MTBDDs only to represent the transition functions, whereas the cited references use them also to represent and manipulate sets of states.

We will make use of the following operations on MTBDDs. Let $f(c_1, \ldots, c_n)$ be a MTBDD defined on the atomic predicates $c_1, \ldots, c_n$ and returning values in some set T. The cofactor operation f_l where l is a literal (i.e., an atom or its negation) is the partial evaluation of f w.r.t. the literal l. This operation is trivially extended to conjunctions of literals (or monomials) m and is noted f_m. The *support* $\mathsf{supp}(f)$ is the set of predicates involved in f. $\mathsf{T}(f) \subseteq T$ denotes the set of *terminals* of f. The *guard* of a terminal t in f is the BDD $\mathsf{guard}(t, f)$ defined by $\mathsf{guard}(t, f)(c_1, \ldots, c_n) \Leftrightarrow (f(c_1, \ldots, c_n) = t)$. $\mathsf{path}(f)$ denotes the set of paths (m, t) of f, where f is the conjunction of literals encountered from the root node of f to the terminal t. In the case of a BDD, $\mathsf{path}(f)$ will denote the set of paths leading to *true*. For a BDD f (a MTBDD with $T = \{true, false\}$), its factorization $\mathsf{fact\_bdd}(f)$ is the conjunction $f = m \wedge f_m$ where m the smallest monomial such that $f \Rightarrow m$.

5 An Overview of Solutions for Abstract Transfer Functions

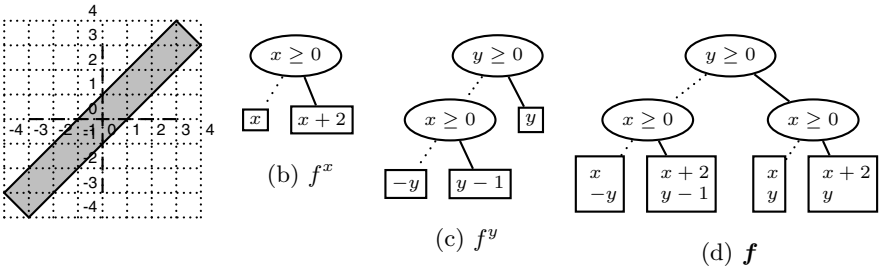
We present here a set of simple methods to approximate transfer functions. We consider a symbolic dynamic system $(V^s, V^i, Init, Ast, (f^v)_{v \in V}, Final)$ generating a discrete dynamic system $(S, I, Init, Ast, \tau, Final)$, where the relation $\tau \subseteq S \times I \times S$ is defined by

$$\tau(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'}) \iff Ast(\boldsymbol{s}, \boldsymbol{i}) \wedge (\boldsymbol{s'} = f(\boldsymbol{s}, \boldsymbol{i})) \iff Ast(\boldsymbol{s}, \boldsymbol{i}) \wedge \bigwedge_{v \in V^s} s'_v = f^v(\boldsymbol{s}, \boldsymbol{i})$$

For any concrete predicate $x \in \wp(D^{\boldsymbol{n}})$, $[\![x]\!]^\alpha \sqsupseteq \alpha(x)$ will denote in the sequel of the paper some correct approximation of x in $A^{(\boldsymbol{n})}$. As we will later use postcondition and precondition on a partitioned abstract domain, we define the abstract transfer functions in a somehow non-standard way. τ is first extended to the powerset $\tau^\gamma : \wp(S \times I \times S) \to \wp(S \times I \times S)$ with $\tau^\gamma(Z) = \{(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'}) \in Z \mid \tau(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'})\}$. Our aim is now to approximate τ^γ with $\tau^\alpha \in A^{(\boldsymbol{n}^s + \boldsymbol{n}^i + \boldsymbol{n}^s)} \to A^{(\boldsymbol{n}^s + \boldsymbol{n}^i + \boldsymbol{n}^s)}$, in order to obtain the following abstract transfer functions:

$$\begin{aligned} \mathrm{post}^\alpha(X, Y) &\triangleq [\![\mathrm{post}(\gamma(X)) \cap \gamma(Y)]\!]^\alpha = \exists \boldsymbol{s} \exists \boldsymbol{i}\ \tau^\alpha(X(\boldsymbol{s}) \sqcap Y(\boldsymbol{s'})) \\ \mathrm{pre}^\alpha(X, Y) &\triangleq [\![\gamma(X) \cap \mathrm{pre}(\gamma(Y))]\!]^\alpha = \exists \boldsymbol{i} \exists \boldsymbol{s'}\ \tau^\alpha(X(\boldsymbol{s}) \sqcap Y(\boldsymbol{s'})) \end{aligned}$$

where $X(\boldsymbol{s})$ and $Y(\boldsymbol{s'})$ are two abstract values.



(a) $X(x, y)$ (b) f^x (c) f^y (d) $\boldsymbol{f}$

Fig. 1. MTBDDs of the transition functions and their product

Best correct upper-approximation. A first solution is to consider the product MTBDD $f = \prod_{v \in V} f^v$, and to consider all of its paths, after having intersected Z with Ast:

$$\tau^{\alpha}_{\text{best}}(Z) = \bigsqcup_{a \in \mathsf{path}(Ast)} \left(\bigsqcup_{(m,\boldsymbol{e}) \in \mathsf{path}(\boldsymbol{f}_a)} [\![a \wedge m]\!]^{\alpha}(\boldsymbol{s}, \boldsymbol{i}) \sqcap [\![\boldsymbol{s'} = \boldsymbol{e}(\boldsymbol{s}, \boldsymbol{i})]\!]^{\alpha} \sqcap Z(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s'}) \right)$$

If we assume that for each variable $v \in V^s$, f^v depends on at most $\#c$ predicates, and we note $\#v = |V^s|$, $\boldsymbol{f}$ have at most $2^{\#c \cdot \#v}$ paths. If we suppose that Ast depends on predicates involved in the f^v, this method requires in the worst case $\mathcal{O}(2^{\#c \cdot \#v})$ atomic operations. We obtain the best correct upper approximation if conjunctions of conditions and elementary functions have best correct approximations, i.e., if $[\![m]\!]^{\alpha} = \alpha([\![m]\!])$ and $[\![\boldsymbol{s'} = \boldsymbol{e}(\boldsymbol{s}, \boldsymbol{i})]\!]^{\alpha} = \alpha([\![\boldsymbol{s'} = \boldsymbol{e}(\boldsymbol{s}, \boldsymbol{i})]\!])$.

Example 3. Let us take $V = V^s = \{x, y\}$, $S = \mathbb{R}^2$, $A = \mathsf{Pol}(\mathbb{R}^2)$, $X = \{x - 1 < y \leq x + 1, -7 \leq x + y \leq 7\}$, $Y = \top$, $Ast = true$ and the transition function defined by:

$$f^x = \textbf{if } x \geq 0 \textbf{ then } x + 2 \textbf{ else } x$$
$$f^y = \textbf{if } y \geq 0 \textbf{ then } y \textbf{ else } (\textbf{if } x \geq 0 \textbf{ then } y - 1 \textbf{ else } -y)$$

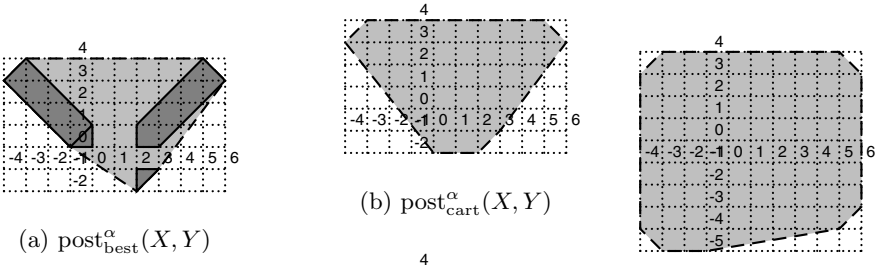
The abstract value X and the MTBDDs of f^x, f^y and their product $\boldsymbol{f}$ are depicted on Fig. 1. We compute

$$\begin{aligned}\text{post}^{\alpha}(X,Y)_{\text{best}} &= \exists x \exists y(\ X \sqcap \{y<0, x<0\} \sqcap \{x'=x, y'=-y\} \ \sqcup\ X \sqcap \{y<0, x \geq 0\} \sqcap \{x'=x+2, y'=y-1\} \ \sqcup \\ &\qquad X \sqcap \{y \geq 0, x<0\} \sqcap \{x'=x, y'=y\} \ \sqcup\ X \sqcap \{y \geq 0, x \geq 0\} \sqcap \{x'=x+2, y'=y\} \\ &= \{y' \leq 4, -1 \leq x'+y' \leq 9, -4 < y'-x' \leq 7, 2x'+3y' \geq -2, 3y'-4x' > -15\}\end{aligned}$$

The result is depicted on Fig. 2.(a). □

For the sake of simplicity, we assume in the remainder of this section that $Ast = true$.

Cartesian product. To avoid the blow-up of paths due to the product of functions $\prod_{v \in V} f^v$, an alternative is to compute separately the effect of each function f^v on the dimension v' of the state space:



(a) $\text{post}^{\alpha}_{\text{best}}(X,Y)$

(b) $\text{post}^{\alpha}_{\text{cart}}(X,Y)$

(c) $\text{post}^{\alpha}_{\text{cart}}(X,Y)$ factorized on $x \geq 0$

(d) $\text{post}^{\alpha}_{\text{cart,abs}}(X,Y)$

The polygons in dark grey are the images of the polygons obtained by intersecting X with the paths of the product MTBDD $\boldsymbol{f}$. Here their set-union corresponds to the concrete image $\text{post}(X)$.

Fig. 2. Image of X by different postcondition operators, with $R = \top$

$$\tau^{\alpha}_{\text{cart}}(Z) = \underset{v \in V}{\sqcap} \left(\bigsqcup_{(m,e^v) \in \mathsf{path}(f^v)} [\![m]\!]^{\alpha}(\boldsymbol{s},\boldsymbol{i}) \sqcap [\![s'_v = e^v(\boldsymbol{s},\boldsymbol{i})]\!]^{\alpha} \sqcap Z(\boldsymbol{s},\boldsymbol{i},\boldsymbol{s'}) \right)$$

Using the same notations, the complexity is here $\mathcal{O}(\#v \cdot 2^{\#c})$. Despite the decomposition into a cartesian product, the result is relational because primed variable may be related through their relationship with unprimed variables. However, it is less precise, because lattices are not distributive in general: we only have $X \sqcap (Y_1 \sqcup Y_2) \sqsupseteq (X \sqcap Y_1) \sqcup (X \sqcap Y_2)$.

Example 4. Taking the example 3, we compute

$$\begin{aligned}
&\text{post}^{\alpha}_{\text{cart}}(X,Y)\\
&= \exists x \exists y \big((X \sqcap \{x<0\} \sqcap \{x'=x\} \sqcup X \sqcap \{x \geq 0\} \sqcap \{x'=x+2\}) \sqcap\\
&\qquad (X \sqcap \{y<0, x<0\} \sqcap \{y'=-y\} \sqcup X \sqcap \{y<0, x \geq 0\} \sqcap \{y'=y-1\} \sqcup X \sqcap \{y \geq 0\} \sqcap \{y'=y\}) \big)\\
&= \{y' \leq 4, -1 \leq x'+y' \leq 9, -4 < y'-x' \leq 7, 2x'+3y' \geq -2, 3y'-4x' > -15\}
\end{aligned}$$

The result is depicted on Fig. 2.(b). □

Abstracting predicates. Cartesian product allows to tackle the complexity of the postcondition operator which comes from the product of the individual functions f^v. However, what to do when the single functions f^v have themselves complex definitions ? Such a situation happens in synchronous programs. A solution is to simply omit the intersection with predicates involved in MTBDDs and BDDs, and to consider only the sets of their terminals. Together with cartesian

decomposition, we obtain:

$$\tau^\alpha_{\text{abs}}(Z) = \bigsqcap_{v\in V} \left(\bigsqcup_{e^v\in\mathsf{T}(f^v)} [\![s'_v = e^v(\boldsymbol{s},\boldsymbol{i})]\!]^\alpha \sqcap Z(\boldsymbol{s},\boldsymbol{i},\boldsymbol{s'}) \right)$$

For the example 3 we get the result depicted on Fig. 2.(d). Assuming that $\#e \leq 2^{\#c}$ is the maximum number of terminals in the MTBDDs, the complexity is here $\mathcal{O}(\#e \cdot \#v)$. In practice we usually have $\#e \ll 2^{\#c}$ and the complexity is considerably reduced. Of course, ignoring predicates garding elementary functions leads to a less precise approximation of τ^γ than $\tau^\alpha_{\text{cart}}$.

Theorem 1 (Precision and complexity). *The different τ^α operators satisfy the inclusion relationship $\tau^\alpha_{\text{best}} \sqsubseteq \tau^\alpha_{\text{cart}} \sqsubseteq \tau^\alpha_{\text{abs}}$ and their complexity is summarized in the following table.*

best	cart	abs
$2^{\#v\cdot\#c}$	$\#v \cdot 2^{\#c}$	$\#v \cdot \#e$

Combination of the above solutions. The possibilities presented above can be refined and combined in order to lead to solutions that exhibit intermediate accuracy and complexity. For the predicate abstraction, an obvious refinement consists in abstracting only a subset of predicates. For the cartesian product, it is possible to partition V into an union of subsets, instead of partitioning it into singletons. For instance, variables which are known to be strongly related should be kept in the same subset in the cartesian decomposition. Also, when the same condition c appears in many functions f^v, it is better to first intersect X with $[\![c]\!]^\alpha$ and $[\![\bar{c}]\!]^\alpha$ before applying cartesian decomposition. This allows to establish relations between variables depending whether the condition c is satisfied or not and makes the result more precise. We say that we *factorize* the test on the condition c.

Example 5. Taking again example 3, we first factorize the test on the predicate $x \geq 0$ and then apply the cartesian decomposition:

$$\begin{aligned}
&\text{post}^\alpha(X,Y)\\
&= \exists x\exists y\Big(\big((X\sqcap\{x{<}0\}\sqcap\{x'{=}x\}) \sqcap (X\sqcap\{x{<}0,y{<}0\}\sqcap\{y'{=}{-}y\} \sqcup X\sqcap\{x{<}0,y{\geq}0\}\sqcap\{y'{=}y\})\big) \sqcup\\
&\qquad \big((X\sqcap\{x{\geq}0\}\sqcap\{x'{=}x{+}2\}) \sqcap (X\sqcap\{x{\geq}0,y{<}0\}\sqcap\{y'{=}y{-}1\} \sqcup X\sqcap\{x{\geq}0,y{\geq}0\}\sqcap\{y'{=}y\})\big)\Big)\\
&= \{y'{\leq}4,{-}1{\leq}x'{+}y'{\leq}9,{-}4{<}y'{-}x'{\leq}7,2x'{+}3y'{\geq}{-}2,3y'{-}4x'{>}{-}15\}
\end{aligned}$$

which is depicted on Fig. 2.(c).

Comparison with existing works. Let us now detail the method of [5] cited in the introduction. In order to represent sets of state and transition functions, [5] uses formulas of the form: $\bigvee_i \bigwedge_{t\in\mathcal{T}} a^t_i$, where the a^t's are formulas of type t. The computation of a postcondition is done by considering the conjunction of the formulas representing respectively the set of states and the transition relation, by putting it in the above-mentioned form, and by eliminating unprimed variables from the obtained formula. The methods [18,19,3] cited in the introduction, instead of the "disjunctive formulas" of [5], uses BDDs, the nodes of

which are labelled by predicates (or constraints). Postcondition computations are performed in a similar way, but with BDD operations, the complexity of which depends on the number of nodes of BDDs.

The obtained postcondition operators are then similar to our best correct approximation method, with the difference that there is no abstract lattice: disjunctions and conjunctions are exact. In both cases, the semantics of predicates is not taken into account by the logical operation on formulas (resp. BDDs) and they can contain unsatisfiable conjunctions (resp. paths). So for emptiness or inclusion test it is necessary to iterates on conjunctions (resp. paths) of the representation and to remove the unsatisfiable ones. In the case of BDDs the complexity of this operation depends obviously on the number of *paths* of the BDD, instead of the number of *nodes*. That means that the sharing provided by BDDs is not so effective here.

6 A Parameterized Heuristic Method

We suggest here a method that combines the previous solutions and offers a parameterized tradeoff between accuracy and efficiency.

6.1 Decomposed Relations and Their Construction

Our abstract relations will be described by a syntaxical structure called *decomposed relation*.

Definition 4 (Decomposed relation). *An* decomposed relation R^V*, indexed by a set* V *of variables, is defined by the following syntax:*

$$
\begin{array}{lll}
ll\ R^V ::= & \mathbf{Assert}(m, R^V) & m = \bigwedge_i c_i \text{ with } \forall i : c_i \in \mathcal{C} \\
| & \mathbf{Ite}(c, R^V, R^V) & c \in \mathcal{C} \\
| & \mathbf{Cart}(R^{V_1}, \ldots, R^{V_p}) & \{V_1, \ldots, V_p\} \text{ being a partition of } V; \\
| & \mathbf{Abs}(v, f) & V = \{v\}, f \in \mathcal{F}
\end{array}
$$

Assert is used to intersect the current value with a conjunction of conditions, **Ite** is the if-then-else operator which allows to *open a test* on a condition, **Cart** is the cartesian decomposition operator, and the terminal construct **Abs** allows either to abstract the predicates guarding elementary operations in a conditional function, either to evaluate the effect of an elementary operation. The semantics of a decomposed relation is defined inductively on the table 1. The figure 3 depicts two decomposed relations, where oval boxes denote **Assert**, triangular boxes **Cart**, and diamond boxes **Ite** operators.

Building a correct decomposed relation. Now that we defined a syntaxical structure to describe abstract relations we need an algorithm that generate a decomposed relation R such that $[\![R]\!]^\alpha$ is a correct approximation of τ^γ. We present here such an algorithm. We first assume that $Ast = true$. The idea of the algorithm is to use the **Cart** and **Ite** constructs to factorize the tests on the

Table 1. Semantics of a decomposed relation

R^V	$[\![\cdot]\!]^\alpha : A^{(2\boldsymbol{n}^s+\boldsymbol{n}^i)} \to A^{(2\boldsymbol{n}^s+\boldsymbol{n}^i)}$
$\mathbf{Assert}(m, R)$	$\lambda X \,.\, [\![R]\!]^\alpha(X \sqcap [\![m]\!]^\alpha)$
$\mathbf{Ite}(c, R_t, R_e)$	$\lambda X \,.\, [\![R_t]\!]^\alpha(X \sqcap [\![c]\!]^\alpha) \sqcup [\![R_e]\!]^\alpha(X \sqcap [\![\bar{c}]\!]^\alpha)$
$\mathbf{Cart}(R_1, \ldots, R_p)$	$\lambda X \,.\, \bigsqcup_{1\leq i\leq p}([\![R_i]\!]^\alpha(X))$
$\mathbf{Abs}(v, f)$	$\lambda X \,.\, \bigsqcup_{e\in\mathsf{T}(f)} X \sqcap [\![s'_v = e(\boldsymbol{s}, \boldsymbol{i})]\!]^\alpha$

Algorithm 1 Decomposing the transition relation

Function *decompose*$(V, \boldsymbol{f}, d)$
Parameter: *An integer* $0 \leq \mathrm{dmax}_{\mathrm{ite}} \leq \infty$ *(max. depth of* **Ite** *constructs)*
Input: *A set of variables* V*, a vector* $\boldsymbol{f}$ *of conditional functions, and an integer* d
Output: *A decomposed relation* R

```
(1)  begin
(2)    if (∀v ∈ V : f^v ∈ ⋃_t E_t^(n_t)) or (d = dmax_ite) then
(3)      return Cart({Abs(v, f^v) | v ∈ V})
(4)    else
(5)      let c = choose_cond(V) in
(6)      let V^i = select_var(c, V) in
(7)      let R+ = decompose(V^i, f_c^{V^i}, d + 1) in
(8)      let R− = decompose(V^i, f_c̄^{V^i}, d + 1) in
(9)      let R^o = decompose(V \ V^i, f^{V\V^i}, d) in
(10)     return Cart(Ite(c, R+, R−), R^o)
(11) end
```

predicates that are involved in several transition functions, and to compute the effect of elementary operations, by using **Cart** and **Abs** operators.

Let us detail the algorithm 1. First of all, in line (1), if all the variables in V are associated to elementary functions, they can be directly computed by decomposing them into singletons and using the **Abs** construct. Also, if the maximum depth $d = \mathrm{dmax}_{\mathrm{ite}}$ of **Ite** constructs has been reached, we use the predicate abstraction, instead of going on opening tests with the **Ite** construct.

Otherwise, we should open a test. Line (5) selects a predicate on which a test will be opened, then line (6) selects the variables that will be involved in the two branches of the test. Once this is done, we apply recursively the algorithm. On one hand, lines (7–8) compute the cofactor of the conditional functions of the variables V^i according to the predicate c and apply recursively the algorithm, in order to generate the two branches of the **Ite** construct. On the other hand, line (9) applies recursively the algorithm to the remaining dimensions. Line (10) puts together the results of the recursive calls.

We have still to define the functions *choose_cond* and *select_var*. Our heuristic to select a condition is to choose the one which appears in the greatest number of conditional functions. The algorithm 2 describe the variable selection algo-

Algorithm 2 Selecting the variables to be put in the **Ite** construct

Function *select_var*$(c, V, \boldsymbol{f})$
Parameter: *An integer* $-1 \leq \mathrm{dmax}_{\mathrm{select}} \leq \infty$
Input: *a condition* c*, a set* V *of variable and the associated vector* $\boldsymbol{f}$ *of conditional functions*
Output: *a subset* W *of* V
(1) **begin**
(2) **if** $\mathrm{dmax}_{\mathrm{select}} = -1$ **then**
(3) **return** $\{v\}$*, such that* $c \in \mathsf{supp}(f^v)$
(4) **else**
(5) **let** $W_c = \{v \in V \mid c \in \mathsf{supp}(f^v)\}$ **in**
(6) **let** $W_o = \{v \in V \mid |\mathsf{supp}(f^v)| < \mathrm{dmax}_{\mathrm{select}}\}$ **in**
(7) **return** $W_c \cup W_o$
(8) **end**

rithm. In the case where $\mathrm{dmax}_{\mathrm{select}} \geq 0$, the line (5) selects all the variables, the associated function of which depends on the condition: it is natural to put these functions in the branches of the **Ite** construct. What to do with the other functions ? If we want for instance to obtain the best correct approximation, we have to select *all the remaining variables*. In order to leave opened a full range of possibilities, we use the parameter $\mathrm{dmax}_{\mathrm{select}}$: if a function depends on less than $\mathrm{dmax}_{\mathrm{select}}$ predicates, line (6) will select the associated variable. The special case $\mathrm{dmax}_{\mathrm{select}} = -1$ allows to forbid any factorization of predicates and to obtain the $\mathrm{post}^{\alpha}_{\mathrm{cart}}$ described in section 5.

Satisfying the assertion. If we remove the assumption that $Ast = true$, we have to satisfy it before computing the relation. This is performed by the algorithm 3, which use **Ite** and **Assert** constructs to recursively decompose the MTBDD of Ast. The parameter $\mathrm{dmax}_{\mathrm{assert}}$ allows to bound the depth of the **Ite** constructs. If the line (7) is executed, then the assertion will be only partially satisfied on the corresponding branch.

Theorem 2 (Correctness). *For any decomposed relation* R *constructed by the algorithm 3,* $[\![R]\!]^{\alpha} \sqsupseteq \tau^{\alpha}_{\mathrm{best}}$

6.2 Complexity and Accuracy Results

Let us denote by $R_{(a,i,s)}$ the decomposed relation generated by the algorithm 3 with the parameters $\mathrm{dmax}_{\mathrm{assert}} = a$, $\mathrm{dmax}_{\mathrm{ite}} = i$, $\mathrm{dmax}_{\mathrm{select}} = s$. First, notice that by adjusting these parameters, it is possible to obtain some of the postcondition operators described in section 5; for instance:

$$[\![R_{(\infty,\infty,\infty)}]\!]^{\alpha} = \tau^{\alpha}_{\mathrm{best}}$$
$$[\![R_{(\infty,\infty,-1)}]\!]^{\alpha} = \tau^{\alpha}_{\mathrm{cart}}$$
$$[\![R_{(0,0,-1)}]\!]^{\alpha} = [\![R_{(0,0,0)}]\!]^{\alpha} = \tau^{\alpha}_{\mathrm{abs}}$$

Algorithm 3 Satisfying the assertion

Function *make*$(a, \boldsymbol{f}, d)$
Parameter: *An integer* $\mathrm{dmax}_{\mathrm{assert}}$
Input: *An assertion* a*, a vector* $\boldsymbol{f}$*, and an integer* d
Output: *A decomposed relation* R

```
(1)  begin
(2)    let (m, a') = factorize_bdd(a) in
(3)    match a' with
(4)    Δ(true) → return Assert(m, decompose(V, f_m, 0))
(5)    Δ(c, a⁺, a⁻) →
(6)      if d = dmax_assert then
(7)        return Assert(m, decompose(V, f_m, 0))
(8)      else
(9)        let R⁺ = make(a⁺, f_{m∧c}, d + 1) in
(10)       let R⁻ = make(a⁻, f_{m∧c̄}, d + 1) in
(11)       return Assert(m, Ite(c, R⁺, R⁻))
(12) end
```

Theorem 3. *The following relationships between decomposed relations holds:*

$$\begin{gathered}[\![R_{(a+1,\infty,\infty)}]\!]^\alpha \sqsubseteq [\![R_{(a,\infty,\infty)}]\!]^\alpha \\ [\![R_{(a,i+1,s)}]\!]^\alpha \sqsubseteq [\![R_{(a,i,s)}]\!]^\alpha \\ [\![R_{(a,\infty,\infty)}]\!]^\alpha \sqsubseteq [\![R_{(a,i,1)}]\!]^\alpha \sqsubseteq [\![R_{(a,i,0)}]\!]^\alpha \sqsubseteq [\![R_{(a,i,-1)}]\!]^\alpha\end{gathered}$$

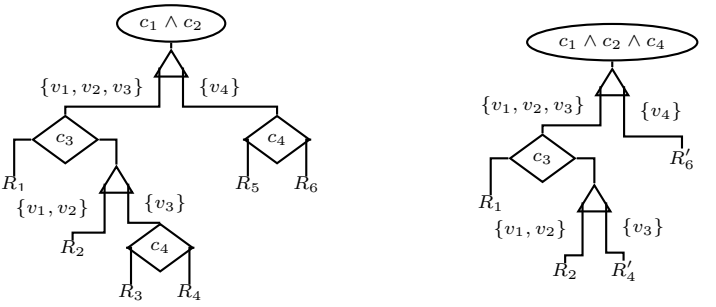
Moreover, the size of $R_{(a,i,s)}$ *is in* $\mathcal{O}(2^{\min(a+d,\#c)} \cdot \#e \cdot \#v)$ *for* $s \leq 1$*, and in* $\mathcal{O}(2^{\min(a+d,\#c+(\#v-1)\cdot\min(s,\#c))} \cdot \#e \cdot \#v)$ *for* $s \geq 2$.

The arguments used in the proof are mainly the monotonicity of the operators defining decomposed relations and the property $(X \sqcap Y_1) \sqcup (X \sqcap Y_2) \sqsubseteq X \sqcap (Y_1 \sqcup Y_2)$. Proving more precise results is difficult, and perhaps impossible, because the modification of one parameter can completely change the structure of the recursive calls in the generation algorithm.

7 Specialization of a Decomposition and Partitioned Systems

We show here how to use the assertion to perform a partial evaluation of a decomposed relation on both origin and destination predicates, and how to take advantage of it when a partitioned domain is used.

Principle. Let us consider a decomposed relation R, two abstract values $P(\boldsymbol{s})$ and $Q(\boldsymbol{s'})$, and let us evaluate $[\![R]\!]^\alpha(P \sqcap Q)$. Consider an $\mathbf{Ite}(c, \ldots)$ construct, and let us note m the conjunction of conditions encountered in the **Assert** and **Ite** constructs from the root to this construct. Suppose that during the recursive evaluation following the table 1, the result of the negative branch of the $\mathbf{Ite}(c, \ldots)$ construct returns $\bot$. That means $\boldsymbol{s} \in P$ and $\boldsymbol{s'} \in Q$ are related only if $m \wedge c$



(a) Before specialization

(b) After specialization

Fig. 3. Specialization of a decomposed relation

is satisfied. We can then strengthen the assertion with this information, and recompute a new decomposition. We obtain then a decomposed relation R wich gives a correct approximation of τ for any $X(\boldsymbol{s}) \sqsubseteq P(\boldsymbol{s})$ and $Y(\boldsymbol{s'}) \sqsubseteq Q(\boldsymbol{s'})$.

Example 6. For instance, on the figure 3(a), if we suppose that this happens on the $\mathbf{Ite}(\mathbf{c_4}, \mathbf{R_5}, \mathbf{R_6})$ construct, we get a new assertion $c_1 \wedge c_2 \wedge c_4$ and we can generate the decomposed relation depicted on the figure 3(b). Here the depth of **Ite** constructs is decreased by 1 above R_4 and R_6, which means that less predicates will be abstracted in R'_4 and R'_6.

The general procedure consists in evaluationg $[\![R]\!]^\alpha$ on $P \sqcap Q$, to compute the disjunction g of all the conjunction of conditions leading to $\bot$, to strengthen the formula Ast with $\neg g$, and to recompute a new decomposed relation.

Interest of the specialization. If we fix the parameters (a, i, s), strengthening the assertion by specializing a decomposed relation $R_{(a,i,s)}$ on $P(\boldsymbol{s}) \sqcap Q(\boldsymbol{s'})$ is very interesting, because the algorithm 1 is applied to simpler conditional functions. As a consequence, the number of abstracted predicates is decreased and the factorization can be more effective. So, with a constant complexity, the accuracy is improved.

It also provides a way to compute necessary conditions for a concrete state in P to lead to a concrete state in Q in an incremental way. We can repeat the process of evaluating $[\![R]\!]^\alpha(P \sqcap Q)$, strengthening the assertion and computing a new decomposition. The iteration will converge because only a finite number of predicates are involved in the conditional functions.

Use of specialization on partitioned systems. On a partitioned abstract lattice, where a set of concrete states is abstracted by a n-uplet of disjoint abstract values [16], specialization can be used by specializing assertions and decomposed relations on edges linking two abstract classes. Consider a finite partition $\mathcal{K} = \{P^{(k)} \in A \mid k \in K\}$ of S into abstract values. For each pair of classes (k_1, k_2) we can consider an assertion $Ast^{k_1 \rightsquigarrow k_2}$ and a decomposed relation $R^{k_1 \rightsquigarrow k_2}$ specialized on $P^{(k_1)}(\boldsymbol{s}) \sqcap P^{(k_2)}(\boldsymbol{s'})$. During forward analysis, postconditions will be computed along edges with $\text{post}^{k_1 \rightsquigarrow k_2} = \lambda X \sqsubseteq$

Table 2. Experiment

N	bool var state/input	num var state/input	numerical constraints	Assertion (nodes,paths)
2	7/3	6/0	23	(15,55)
3	10/4	8/0	34	(26,239)
4	13/5	10/0	45	(45,1055)

Conditional functions (for each state variable):
$\approx$ 10 nodes, $\approx$ 20 paths

(a) Program statistics

N	2			3			4		
(a,i,s)	time	ana	ite	time	ana	ite	time	ana	ite
$(1,1,2)$	3.9	1	6	12.0	1	9	107.6	1	11
$(2,2,2)$	6.2	2	25	15.6	2	39	34.4	2	52
$(3,3,2)$	9.0	2	53	28.7	2	118	64.0	2	157
$(5,5,2)$	14.8	2	119	65.9	2	276	208.9	2	588
$(\infty,\infty,2)$	13.9	2	138	75.1	2	478	359.2	2	1424

(b) Property 2, lower bound

N	2		3		4	
(a,i,s)	time	ana	time	ana	time	ana
$(1,1,2)$	failure		failure		failure	
$(2,2,2)$	22.6	4	93.2	4	220.0	4
$(3,3,2)$	31.0	4	136.6	4	479.2	4
$(5,5,2)$	40.2	4	212.4	4	858.3	4
$(\infty,\infty,2)$	43.3	4	291.0	4	swap !	–

(c) Property 2, upper bound

$P^{(k)}.[\![R]\!]^\alpha(X(\boldsymbol{s}) \sqcap P^{(k_2)}(\boldsymbol{s'}))$, and similar precondition operators will be used for backward analysis.

Assertions associated to edges can also be used to refine a partition. Indeed, comparing two assertions $Ast^{k \rightsquigarrow k_1}$ and $Ast^{k \rightsquigarrow k_2}$ allows to separate those states in $P^{(k)}$ wich can only lead to k_1 or k_2. Such information is the basis of several partition refinement techniques, in abstract interpretation [16,11] or in the combination of deduction and model-checking [24].

Last but not least, our technique allows to link the precision of the transfer functions to the precision induced by the partition: when a partition is refined, specialization becomes stronger and decomposed relations are more precise.

8 Experimental Evaluation

We implemented our technique in the tool NBAC [16], which perform verification of LUSTRE programs [13] having both Boolean and numerical variables. The algorithms implemented in this tool include forward and backward analysis, automatic partition refinement and interleaved iterations of these.

We experimented it on a railway control system described in [16]. Table 2(a) gives information on the input program, where N is the number of trains. The assertion, which models the environment and from which comes in a great part the complexity of the system, states basically that the trains obey the orders they are given. Tables 2(b) and 2(c) gives the verification results for the property 2 defined in [16]. We give for different values of the parameters the time in seconds, the number of analysis cycles followed by partition refinement cycles, and the average number of **Ite** per decomposed relation (only in table 2(b)). We used the standard Boolean lattice for the Boolean variables (i.e., Boolean variables are not abstracted) and the convex polyhedra lattice $\mathsf{Pol}(\mathbb{Q}^n)$ for the numerical variables. We were able to verify the property only for $s = \mathrm{dmax}_{\mathrm{select}} \geq 2$. The

tables shows that on this example a small bound on the maximum depth of nested **Ite** operators still allows the verification to be successful, and at a very cheap cost compared to the cost observed when this depth is not bounded (row indexed by $(\infty, \infty, 2)$). Observe also that the use of less precise operators does not require a higher number of refinement steps to prove the property, which shows that a greater precision is not needed for the proof.

9 Conclusion

We proposed a generic framework to combine different datatypes in Abstract Interpretation of reactive systems. Our solution is based on the assumption that transition functions of the analysed system can be represented by conditional functions as defined in definition 2. This quite liberal assumption allows us to design a generic method to analyse and verify such programs, which relies on the following ingredients:

- Combining abstract lattices devoted to each datatype into a global abstract lattice with the very simple *direct product* operator;
- Using *partitioning* on the previous lattice to improve its precision;
- Using *decomposed relations* to compute transfer functions and to adapt their precision with the precision induced by the partitioning.

The paper focuses more precisely on the last point. Our preliminary experiments shows the usefulness of approximating transfer functions in a flexible and parametrized way: it can make the global method considerably more efficient. Our future plans are to experiment the combination of other datatypes and/or abstract lattices with this method.

References

1. P. Aziz Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Computer Aided Verification, CAV'98*, volume 1427 of *LNCS*, July 1998.
2. R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, April 1997.
3. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer Aided Verification, CAV'99*, volume 1633 of *LNCS*, July 1999.
4. B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Computer Aided Verification, CAV'96*, volume 1102 of *LNCS*, July 1996.
5. T. Bultan, R. Gerber, and C. League. Composite model checking: Verification with type-specific symbolic representations. *ACM Transactions on Software Engineering and Methodology*, 9(1), January 2000.
6. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Computer Aided Verification, CAV'97*, volume 1254 of *LNCS*, June 1997.

7. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combination of abstract domains for logic programming: open product and generic pattern construct. *Science of Computer Programming*, 38, 2000.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM Symposium on Principles of Programming Languages, POPL'79*, San Antonio, January 1979.
10. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 1992.
11. R. Giacobazzi and E. Qintarelli. Incompleteness, counterexamples, and refinement in abstract model-checking. In *Static Analysis Symposium, SAS'01*, volume 2126 of *LNCS*, July 2001.
12. N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.
13. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
14. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2), August 1997.
15. T. Henzinger, P.-H. Ho, and H. Wong-Toï. HyTech: A Model Checker for Hybrid Systems. In *Computer Aided Verification, CAV'97*, number 1254 in LNCS, June 1997.
16. B. Jeannet. Dynamic partitioning in linear relation analysis. Application to the verification of reactive systems. *Formal Methods in System Design*. 40 pages, to appear, available as a BRICS Research Report `http://www.brics.dk/RS/00/38`.
17. B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99*, volume 1694 of *LNCS*, Venezia (Italy), September 1999.
18. C. Mauras. Calcul symbolique et automates interprétés. Technical Report 10, IRCyN, November 1996.
19. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Computer Science Logic*, The IT University of Copenhagen, Denmark, September 1999.
20. G. Nelson and C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2), 1979.
21. F. Nielson. Tensor products generalize the relational data flow analysis method. In *Fourth Hungarian Computer Science Conference*, 1985.
22. F. Nielson and H. R. Nielson. The tensor product in wadler's analysis of list. *Science of Computer Programming*, 22, 1994.
23. R. Shostak. Deciding combination of theories. *Journal of the ACM*, 31(1), 1984.
24. H. Sipma, T. Uribe, and Z. Manna. Model checking and deduction for infinite-state systems. In *Logical Perspectives on Language and Information*. CSLI publications, 2001.
25. T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01*, volume 2031 of *LNCS*, April 2001.