Prototypes II

Deliverable D4-5

ANR project VECOLIB

March 2017

Abstract

This deliverable presents three prototypes developed during the Vecolib project: (i) a constraint solver for sets and multisets, (ii) an abstract domain library for containers, and (iii) an implementation of an alternating data automata emptiness checker.

1 A Decision Procedure for a Theory of Sets and Multisets

1.1 Motivation and related work

Several containers are implementations of set or multiset mathematical structures. Therefore, the specification of these containers are expressed in terms of set and multiset constraints. For example, an abstract description of an `add` operation that inserts a value `e` into a data structure is then:

```
Model(add(e, t)) = {e} U Model(t)
```

where `Model` denotes an abstraction function mapping the data structure values into the set of elements stored in the value. Other variants of the specification can use multisets or lists instead of sets.

Another way to express the abstraction of the data structure implementing the container is using a inductively defined refinement relation. For example, the inductive definition of a binary search tree storing integer values rooted at E may be transformed into a refinement relation by adding to its parameters the multiset of values M as follows (the logic used is an extension of Separation Logic):

$$\mathtt{bst}(E,M) ::= E = nil \land M = [\![]\!] \quad (1)$$

$$\begin{aligned}\mathtt{bst}(E,M) ::= {} & \exists L,R,d.\ E \mapsto \{(\mathtt{left},L),(\mathtt{right},R),(\mathtt{data},d)\} \\ & \star\, \mathtt{bst}(L,M_L) \star \mathtt{bst}(R,M_R) \\ & \land E \neq nil \land M_L \leq [\![d]\!] < M_R \land M = M_L \oplus [\![d]\!] \oplus M_R\end{aligned} \quad (2)$$

The first equation specifies the case of an empty tree, where the root E is null and the parameter M is an empty bag. The second equation specifies a non null node at location E where the fields `left`, `right`, and `data` contain as values L, R, resp. d (atom $E \mapsto \{(\mathtt{left},L),(\mathtt{right},R),(\mathtt{data},d)\}$). The locations L and R are roots of binary search trees of values in M_L resp. M_R (atoms $\mathtt{bst}(L,M_L)$ resp. $\mathtt{bst}(R,M_R)$), disjoint pairwise and from the node at E (due to the use of the separation conjunction $\star$). The constraint between the multi-sets of values and the value d, $M_L \leq [\![d]\!] < M_R$, specifies that all values in the left sub-tree are less than d and d is strictly less than all values in the right sub-tree.

To verify the correctness of the programs using these containers or the implementation of the containers, the verification tools (analysers or proof systems) need decision procedures for checking the satisfiability of set and multiset constraints. Notice that, in general, the theory of sets in undecidable. Hopefully, the fragments used in the specification of container library does not include the full theory of sets or multisets and need only the unquantified or existentially unquantified fragment with few operations.

Powerful decision procedures have been proposed and are available for sub-theories of set and multiset. For example, SMT solvers like CVC4 [1] include support for the theory of finite sets. Moreover, the SMT-lib standard [2] is ready to integrate such a theory (for lists and map also).

The existing decision procedure may be classified into three classes depending on the principle used:

- by reduction of the basic operations of finite sets and maps to McCarthys theory of arrays, which is supported by existing solvers, e.g., Z3 [3];
- by reduction to algebraic datatypes that are supported by many solvers, e.g., CVC4;
- by reduction to the boolean algebra with with Presburger arithmetic (BAPA), used for example is GRASS solver.

1.2 A new solver for set and bag constraints

Motivated by our work on implementation of containers, we considered a sub-theory of set and multiset constraints which is used to specify refinement relations like `bst` and may be used for more general specification, like add above. The multi-set constraints include classical atoms, like multi-set inclusion or membership, as well as constraints ordering multi-sets with respect to their elements or constraining the minimum or maximum of a multi-set. It extends the set of constraints allowed by CVC4 for sets of integers, but it is limited to integer elements.

The decision procedure is based on the rewriting of a multi-set formula φ into an equi-satisfiable formula $\tilde{\varphi}$ in quantifier free logic of linear arithmetics. The latter logic is called QFLIA theory in SMT-lib format [2] and it is the input theory of very efficient solvers, e.g., CVC4 or Z3. Moreover, because these solvers support also an extension of this theory with uninterpreted functions, we provide an alternative rewriting of φ into this theory.

We present here only the logic fragment dealt by our solver, i.e., the fragment of *quantifier free logic over bags of integers and linear arithmetics*, QFBILIA. The decision procedure is detailed in the appendix report. The code of this solver is available on github.

We denote by $\bot$, $\top$ the extremum term such as $\forall k \in \mathbb{Z}, k > \bot$, resp. $k < \top$, we have $\bot < \top$ [8]. We donote by $\mathbb{Z}^{\prec}$ the set $\mathbb{Z} \cup \{\bot, \top\}$. We use the classic notations for operations over $\mathbb{Z}$. Integer constants are denoted by k, natural ones by n. Let $V_{int} = \{a, b, c, \ldots\}$ be a finite set of symbols denoting integer variables, i.e., variables with values in $\mathbb{Z}$. Let $V_{ext} = \{m, n, p, \ldots\}$ be a finite set of symbols denoting extremum variables, i.e., variables with values in $\mathbb{Z}^{\prec}$. We denote by $\mathbb{M}[\mathbb{Z}]$ the domain of bags over integers, i.e., the set of functions $\mathbb{Z} \to \mathbb{N}$. Let $V_{bag} = \{x, y, z, \ldots\}$ be a finite set of symbols denoting bag variables, i.e., variables with values in $\mathbb{M}[\mathbb{Z}]$. We suppose that V_{int}, V_{ext} and V_{bag} are disjoint and we don't write explicitly the type ($\mathbb{Z}$, $\mathbb{Z}^{\prec}$ or $\mathbb{M}[\mathbb{Z}]$) of each variable.

Definition 1 *A QFBILIA formula F is defined by the following grammar:*

$$
\begin{array}{rll}
F & ::= L \mid F \vee F \mid F \wedge F \mid \neg F \mid F \Rightarrow F & \textit{formula} \\
L & ::= L_{int} \mid L_{bag} \mid L_{mix} \mid L_{ext} & \textit{boolean atom} \\
L_{int} & ::= T_{int} = T_{int} \mid T_{int} \neq T_{int} \mid T_{int} < T_{int} \mid T_{int} \geq T_{int} & \\
L_{ext} & ::= T_{ext} = T_{ext} \mid T_{ext} \neq T_{ext} \mid T_{ext} < T_{ext} \mid T_{ext} \geq T_{ext} & \\
L_{bag} & ::= T_{bag} = T_{bag} \mid T_{bag} \neq T_{bag} \mid T_{bag} \subseteq T_{bag} \mid T_{bag} \nsubseteq T_{bag} \mid & \\
 & \quad T_{bag} < T_{bag} \mid T_{bag} \geq T_{bag} & \\
L_{mix} & ::= a \in T_{bag} \mid a \notin T_{bag} \mid a \in^n T_{bag} \mid a \notin^n T_{bag} \mid & \\
 & \quad m \in T_{bag} \mid m \notin T_{bag} \mid m \in^n T_{bag} \mid m \notin^n T_{bag} & \\
T_{int} & ::= k \mid a \mid T_{int} + T_{int} \mid T_{int} - T_{int} \mid & \textit{integer term} \\
 & \quad \max(T_{int}, T_{int}) \mid \min(T_{int}, T_{int}) \mid \mathsf{ite}((, F, ,)T_{int}, T_{int}) & \\
T_{ext} & ::= k \mid m \mid \min(T_{bag}) \mid \max(T_{bag}) \mid \mathsf{ite}((, F, ,)T_{ext}, T_{ext}) & \textit{extremum term} \\
T_{bag} & ::= [\![]\!] \mid [\![a]\!] \mid [\![m]\!] \mid x \mid T_{bag} \cup T_{bag} \mid T_{bag} \cap T_{bag} \mid & \textit{bag term} \\
 & \quad T_{bag} \setminus T_{bag} \mid T_{bag} \uplus T_{bag} \mid \mathsf{ite}((, F, ,)T_{bag}, T_{bag}) &
\end{array}
$$

We denote by $\mathcal{F}$ the set of formulas in QFBILIA, by $\mathcal{T}_{int}$ the set of integer terms, by $\mathcal{T}_{ext}$ the set of extremum terms, and by $\mathcal{T}_{bag}$ the set of multi-set terms.

We provide below some examples of constraints in the QFBILIA fragment.

If an integer a is in a multi-set x, the multi-set $[\![a]\!]$ is included in the union of multi-sets x and y:

$$(a \in x) \Rightarrow ([\![a]\!] \subseteq (x \cup y)) \tag{3}$$

If 1 is the smallest element of a multi-set x then x is less than y otherwise x is greater than z:

$$((\min(x) = 1) \Rightarrow (x < y)) \wedge ((\min(x) \neq 1) \Rightarrow (x < z)) \tag{4}$$

If the maximum of a multi-set x is 0 then x is the empty bag or the minimum of x is not 0:

$$(\max(x) = 0) \Rightarrow ((x = [\![]\!]) \vee (\min(x) \neq 0)) \tag{5}$$

2 Abstract Domains for Containers

We proposed in the deliverable D2.2 two abstract domains for the analysis of containers. The first abstract domain, the sequence domain, is designed for the analysis of client programs using position dependent containers like vectors and lists. The second abstract domain, the multiset domain, is designed for programs using value dependent containers.

In this deliverable, we report the details of the implementation of these domains as Ocaml modules that can be used by the abstract interpretation engine of Frama-C.

The analysis of client programs written in C is applied only to programs using the container like an abstract data types whose operations are given in Tables 1 and 2.

Abstract domain for sequences: We recall from the deliverable D2.2 that an abstract value of this domain is a constraint of the form:

$$(A \wedge E \wedge \bigwedge_{G(\mathbf{y}) \in \mathcal{G}} \forall \mathbf{y}.\, G(\mathbf{y}) \Rightarrow U(\mathbf{y})), \text{ where}$$

Table 1: Operations for position dependent containers

`init(c)`	creates the empty container
`free(c)`	deletes the content of the container
`is_empty(c)`	test for empty container
`add_first(c,v)`	push a value before the first position
`add_last(c,v)`	push a value after the last position
`remove_first(c)`	remove the value at the first position
`remove_last(c)`	remove the value at the last position
`first_pos(c)`	return the first position or -1
`next_pos(c,p)`	return the next position or -1
`last_pos(c)`	return the last position or -1
`get_at(c,p)`	return the value at a valid positon or nil
`isin(c,v)`	returns the position of the value in the container or nil

Table 2: Operations for value dependent containers

`init(c)`	creates the empty container
`free(c)`	deletes the content of the container
`is_empty(c)`	test for empty container
`add(c,v)`	push a value
`remove(c,v)`	remove a value
`isin(c,v)`	return true iff the value is in the container

- A encodes aliasing between containers. For its implementations, we use an union-find data structure.
- E constrains the non-container variables as well as the container lengths and positions. We employ the octagon abstract domain from Apron Library [6] of numerical domains.
- the set of universal constraints is implemented by a map from container variables representative in A (i.e., one variable by alias set) to an array of fixed length indexed by the guard patterns in $\mathcal{G}$ and having as elements numerical abstract values representing the constraint $U(\mathbf{y})$.

To deal with several aliasing relation at a program point, we consider a power set domain, where each aliasing constraint is mapped to the corresponding existential and universal constraint.
Abstract domain for mutisets: We recall from the deliverable D2.2 that an abstract value of this domain is a constraint of the form:

$$A \wedge E \wedge \bigwedge_{c \in C} \forall y \in \texttt{positions}(c).\ U(\mathbf{y}) \quad \wedge \quad (\bigwedge_i t_1^i \# t_2^i),\ \text{where}$$

- A encodes aliasing between containers. For its implementations, we use an union-find data structure.
- E constrains the non-container variables as well as the container lengths. We employ the octagon abstract domain from Apron Library [6] of numerical domains.

- the set of universal constraints is implemented by a map from container variables representative in A (i.e., one variable by alias set) to a numerical abstract value (usually in Octagon domain) representing the constraint $U(\mathbf{y})$.
- the set of constraints $t_1^i = t_2^i$ is represented by a a value in the Polyhedra abstract domain.
- the set of constraints $t_1^i \leq t_2^i$ is represented by a value in the Octagon abstract domain, where each subterm of t_1^i is compared with each subterm of t_2^i; the multiset terms $\mathtt{ms}_c$ are represented by one value for every values inside the multiset.

Like for sequences, we built a power-set domain from these values in order to deal with several aliasing relation at a program point.

3 Alternating Automata Modulo Theories

In the rest of this section we fix an interpretation $\mathcal{I}$ and a finite alphabet Σ of *input events*. Given a finite set $\mathbf{x} \subset \mathsf{Var}$ of variables of sort Data, let $\mathbf{x} \mapsto \mathsf{Data}^{\mathcal{I}}$ be the set of valuations of the variables $\mathbf{x}$ and $\Sigma[\mathbf{x}] = \Sigma \times (\mathbf{x} \mapsto \mathsf{Data}^{\mathcal{I}})$ be the set of *data symbols*. A *data word* (word in the sequel) is a finite sequence $(a_1, \nu_1)(a_2, \nu_2)\ldots(a_n, \nu_n)$ of data symbols, where $a_1, \ldots, a_n \in \Sigma$ and $\nu_1, \ldots, \nu_n : \mathbf{x} \to \mathsf{Data}^{\mathcal{I}}$ are valuations. We denote by ε the empty sequence, by Σ^* the set of finite sequences of input events and by $\Sigma[\mathbf{x}]^*$ the set of data words over $\mathbf{x}$.

This definition generalizes the classical notion of words from a finite alphabet to the possibly infinite alphabet $\Sigma[\mathbf{x}]$. Clearly, when $\mathsf{Data}^{\mathcal{I}}$ is sufficiently large or infinite, we can map the elements of Σ into designated elements of $\mathsf{Data}^{\mathcal{I}}$ and use a special variable to encode the input events. However, keeping Σ explicit in the following simplifies several technical points below, without cluttering the presentation.

Given sets of variables $\mathbf{b}, \mathbf{x} \subset \mathsf{Var}$ of sort Bool and Data, respectively, we denote by $\mathsf{Form}(\mathbf{b}, \mathbf{x})$ the set of formulae ϕ such that $\mathrm{FV}^{\mathsf{Bool}}(\phi) \subseteq \mathbf{b}$ and $\mathrm{FV}^{\mathsf{Data}}(\phi) \subseteq \mathbf{x}$. By $\mathsf{Form}^+(\mathbf{b}, \mathbf{x})$ we denote the set of formulae from $\mathsf{Form}(\mathbf{b}, \mathbf{x})$ in which each boolean variable occurs under an even number of negations.

An *alternating data automaton* (ADA or automaton in the sequel) is a tuple $\mathcal{A} = \langle \mathbf{x}, Q, \iota, F, \Delta \rangle$, where:

- $\mathbf{x} \subset \mathsf{Var}$ is a finite set of variables of sort Data,
- $Q \subset \mathsf{Var}$ is a finite set of variables of sort Bool (*states*),
- $\iota \in \mathsf{Form}^+(Q, \emptyset)$ is the *initial configuration*,
- $F \subseteq Q$ is a set of *final states*, and
- $\Delta : Q \times \Sigma \to \mathsf{Form}^+(Q, \overline{\mathbf{x}} \cup \mathbf{x})$ is a *transition function*,

where $\overline{\mathbf{x}} = \{\overline{x} \mid x \in \mathbf{x}\}$. In each formula $\Delta(q, a)$ describing a transition rule, the variables $\overline{\mathbf{x}}$ track the previous and $\mathbf{x}$ the current values of the variables of $\mathcal{A}$. Observe that the initial values of the variables are left unconstrained, as the initial configuration does not contain free data variables. The size of $\mathcal{A}$ is defined as $|\mathcal{A}| = |\iota| + \sum_{(q,a) \in Q \times \Sigma} |\Delta(q, a)|$.

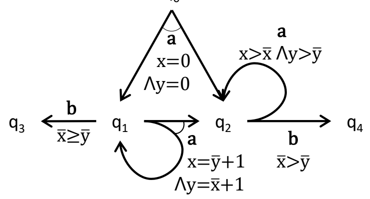

Figure 1: Alternating Data Automaton Example

Example Figure 1 depicts an ADA with input alphabet $\Sigma = \{a,b\}$, variables $\mathbf{x} = \{x,y\}$, states $Q = \{q_0,q_1,q_2,q_3,q_4\}$, initial configuration q_0, final states $F = \{q_3,q_4\}$ and transitions:

$$\begin{array}{rcl}
\Delta(q_0,a) & \equiv & q_1 \wedge q_2 \wedge x \approx 0 \wedge y \approx 0 \\
\Delta(q_1,a) & \equiv & q_1 \wedge q_3 \wedge x \approx \overline{y}+1 \wedge y \approx \overline{x}+1 \\
\Delta(q_1,b) & \equiv & q_3 \wedge \overline{x} \geq \overline{y} \\
\Delta(q_2,a) & \equiv & q_2 \wedge x > \overline{x} \wedge y > \overline{y} \\
\Delta(q_2,b) & \equiv & q_4 \wedge \overline{x} > \overline{y}
\end{array}$$

The missing rules, such as $\Delta(q_0,b)$, are assumed to be $\bot$. Rules $\Delta(q_0,a)$ and $\Delta(q_1,a)$ are universal and there are no existential nondeterministic rules. Rules $\Delta(q_1,a)$ and $\Delta(q_2,a)$ compare past $(\overline{x},\overline{y})$ with present (x,y) values, $\Delta(q_0,a)$ constrains the present and $\Delta(q_1,b)$, $\Delta(q_2,b)$ the past values, respectively. □

Formally, let $\mathbf{x}_k = \{x_k \mid x \in \mathbf{x}\}$, for any $k \geq 0$, be a set of time-stamped variables. For an input event $a \in \Sigma$ and a formula ϕ, we write $\Delta(\phi,a)$ (respectively $\Delta^k(\phi,a)$) for the formula obtained from ϕ by simultaneously replacing each state $q \in \mathrm{FV}^{\mathsf{Bool}}(\phi)$ by the formula $\Delta(q,a)$ (respectively $\Delta(q,a)[\mathbf{x}_k/\overline{\mathbf{x}},\mathbf{x}_{k+1}/\mathbf{x}]$, for $k \geq 0$). Given a word $w = (a_1,\nu_1)(a_2,\nu_2)\ldots(a_n,\nu_n)$, the *run* of $\mathcal{A}$ over w is the sequence of formulae:

$$\phi_0(Q) \Rightarrow \phi_1(Q,\mathbf{x}_0 \cup \mathbf{x}_1) \Rightarrow \ldots \Rightarrow \phi_n(Q,\mathbf{x}_0 \cup \ldots \cup \mathbf{x}_n)$$

where $\phi_0 \equiv \iota$ and, for all $k \in [1,n]$, we have $\phi_k \equiv \Delta^k(\phi_{k-1},a_k)$. Next, we slightly abuse notation and write $\Delta(\iota,a_1,\ldots,a_n)$ for the formula $\phi_n(\mathbf{x}_0,\ldots,\mathbf{x}_n)$ above. We say that $\mathcal{A}$ *accepts* w iff $\mathcal{I},\nu \models \Delta(\iota,a_1,\ldots,a_n)$, for some valuation ν that maps: (1) each $x \in \mathbf{x}_k$ to $\nu_k(x)$, for all $k \in [1,n]$, (2) each $q \in \mathrm{FV}^{\mathsf{Bool}}(\phi_n) \cap F$ to $\top$ and (3) each $q \in \mathrm{FV}^{\mathsf{Bool}}(\phi_n) \setminus F$ to $\bot$. The language of $\mathcal{A}$ is the set $L(\mathcal{A})$ of words from $\Sigma[\mathbf{x}]^*$ accepted by $\mathcal{A}$.
Example The following sequence is a non-accepting run of the ADA from Figure 1 on the word $(a,\langle 0,0\rangle),(a,\langle 1,1\rangle),(b,\langle 2,1\rangle)$, where $\mathsf{Data}^{\mathcal{I}} = \mathbb{Z}$ and the function symbols have standard arithmetic interpretation:

$$\begin{array}{l}
q_0 \overset{(a,\langle 0,0\rangle)}{\Longrightarrow} q_1 \wedge q_2 \wedge x_1 \approx 0 \wedge y_1 \approx 0 \overset{(a,\langle 1,1\rangle)}{\Longrightarrow} \\
\underbrace{q_1 \wedge q_2 \wedge x_2 \approx y_1 + 1 \wedge y_2 \approx x_1 + 1}_{q_1} \wedge \underbrace{q_2 \wedge x_2 > x_1 \wedge y_2 > y_1}_{q_2} \\
\wedge\ x_1 \approx 0 \wedge y_1 \approx 0 \overset{(b,\langle 2,1\rangle)}{\Longrightarrow} \underbrace{q_3 \wedge x_2 \geq y_2}_{q_1} \wedge \underbrace{q_4 \wedge x_2 > y_2}_{q_2} \wedge x_2 \approx y_1 + 1 \\
\wedge\ y_2 \approx x_1 + 1 \wedge \underbrace{q_4 \wedge x_2 > y_2}_{q_2} \wedge x_2 > x_1 \wedge y_2 > y_1 \wedge x_1 \approx 0 \wedge y_1 \approx 0\ \square
\end{array}$$

Here we tackle the following problems:

1. *boolean closure*: given automata $\mathcal{A}_1$ and $\mathcal{A}_2$, both with the same set of variables $\mathbf{x}$, do there exist automata $\mathcal{A}_\cup$, $\mathcal{A}_\cap$ and $\overline{\mathcal{A}_1}$ such that $L(\mathcal{A}_\cup) = \mathcal{A}_1 \cup \mathcal{A}_2$, $L(A_\cap) = \mathcal{A}_1 \cap \mathcal{A}_2$ and $L(\overline{\mathcal{A}_1}) = \Sigma[\mathbf{x}]^* \setminus L(\mathcal{A}_1)$?
2. *emptiness*: given an automaton $\mathcal{A}$, is $L(\mathcal{A}) = \emptyset$?

The first problem has a positive answer. Moreover, the construction of boolean closure automata is effective and takes linear time in the size of the input. Given a set Q of boolean variables and a set $\mathbf{x}$ of variables of sort Data, for a formula $\phi \in \mathsf{Form}^+(Q,\mathbf{x})$, with no negated occurrences of the boolean variables, we define the formula $\overline{\phi} \in \mathsf{Form}^+(Q,\mathbf{x})$ recursively on the structure of ϕ:

$$\begin{array}{lcllcl}
\overline{\phi_1 \vee \phi_2} & \equiv & \overline{\phi_1} \wedge \overline{\phi_2} & \overline{\phi_1 \wedge \phi_2} & \equiv & \overline{\phi_1} \vee \overline{\phi_2} \\
\overline{\neg\phi} & \equiv & \neg\overline{\phi} \text{ if } \phi \text{ not atom} & \overline{\phi} & \equiv & \phi \text{ if } \phi \in Q \\
\overline{\phi} & \equiv & \neg\phi \text{ if } \phi \notin Q \text{ atom} & & &
\end{array}$$

We have $|\overline{\phi}| = |\phi|$, for every formula $\phi \in \mathsf{Form}^+(Q,\mathbf{x})$.

In the following let $\mathcal{A}_i = \langle \mathbf{x}, Q_i, \iota_i, F_i, \Delta_i \rangle$, for $i = 1,2$, where w.l.o.g. we assume that $Q_1 \cap Q_2 = \emptyset$. We define:

$$\begin{array}{lcl}
\mathcal{A}_\cup & = & \langle \mathbf{x}, Q_1 \cup Q_2, \iota_1 \vee \iota_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2 \rangle \\
\mathcal{A}_\cap & = & \langle \mathbf{x}, Q_1 \cup Q_2, \iota_1 \wedge \iota_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2 \rangle \\
\overline{\mathcal{A}_1} & = & \langle \mathbf{x}, Q_1, \overline{\iota_1}, Q_1 \setminus F_1, \overline{\Delta_1} \rangle
\end{array}$$

where $\overline{\Delta_1}(q,a) \equiv \overline{\Delta_1(q,a)}$, for all $q \in Q_1$ and $a \in \Sigma$. The following lemma shows the correctness of the above definitions:

Lemma 1 *Given automata $\mathcal{A}_i = \langle \mathbf{x}, Q_i, \iota_i, F_i, \Delta_i \rangle$, for $i = 1,2$, such that $Q_1 \cap Q_2 = \emptyset$, we have $L(\mathcal{A}_\cup) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$, $L(\mathcal{A}_\cap) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$ and $L(\overline{\mathcal{A}_1}) = \Sigma[\mathbf{x}]^* \setminus L(\mathcal{A}_1)$.*

Proof: See [5]. □

On the other hand, the emptiness problem is undecidable. To this end, we provide a semi-algorithm based on abstraction refinement, given below. We consider a total alphabetical order $\prec$ on Σ and lift it to the total lexicographical order $\prec^*$ on Σ^*. A node $n \in N$ is *covered* if $(n,p) \in \lhd$ or it has an ancestor m such that $(m,p) \in \lhd$, for some $p \in N$. A node n is *closed* if it is covered, or $\Lambda(n) \not\models \Lambda(m)$ for all $m \in N$ such that $\lambda(m) \prec^* \lambda(n)$.

The execution of Algorithm 1 consists of three phases[1]: *close*, *refine* and *expand*. Let n be a node removed from the worklist at line 4. If $\mathsf{Acc}_{\mathcal{A}}(\lambda(n))$ is satisfiable, the counterexample $\lambda(n)$ is feasible, in which case a model of $\mathsf{Acc}_{\mathcal{A}}(\lambda(n))$ is obtained and a word $w \in L(\mathcal{A})$ is returned. Otherwise, $\lambda(n)$ is a spurious counterexample and the procedure enters the refinement phase (lines 11-18). The interpolant for $\Theta(\lambda(n))$ is used to strenghten the labels of all the ancestors of n, by conjoining the formulae of the interpolant to the existing labels.

In this process, the nodes on the path between `r` and n, including n, might become eligible for coverage, therefore we attempt to close each ancestor of n that is impacted by the refinement (line 18). Observe that, in this case the call to Close must uncover each node which is covered by a successor of n (line 4 of the Close function). This is required because, due to the over-approximation of the sets of reachable configurations, the covering relation is not transitive, as explained in [7]. If Close adds a covering edge (n_i, m) to $\lhd$, it does not have to be called for the successors of n_i on this path, which is handled via the boolean flag b.

Finally, if n is still uncovered (it has not been previously covered during the refinement phase) we expand n (lines 20-26) by creating a new node for each successor s via the input event $a \in \Sigma$ and inserting it into the worklist.

Example We show the execution of Algorithm 1 on the automaton from Figure 1. Initially, the procedure fires the sequence a, whose endpoint is labeled with $\top$, in Figure 2 (a). Since this node is uncovered, we check the spuriousness of the counterexample a and refine the label of the node to q_1. Since the node is still uncovered, two successors, labeled with $\top$ are computed, corresponding to the sequences aa and ab, in Figure 2 (b). The spuriousness check for aa yields the interpolant

[1]Corresponding to the Close, Refine and Expand in [7].

Algorithm 1 Impact for ADA Emptiness

```
input: an ADA $\mathcal{A} = \langle \mathbf{x}, Q, \iota, F, \Delta \rangle$ over the alphabet $\Sigma$ of input events
output: true if $L(\mathcal{A}) = \emptyset$ and a data word $w \in L(\mathcal{A})$ otherwise
1: let $\mathcal{T} = \langle N, E, \mathtt{r}, \Lambda, R, T, \lhd \rangle$ be an ART
2: initially $N = E = T = \lhd = \emptyset$, $\Lambda = \{(\mathtt{r}, \iota)\}$, $R = \mathsf{FV}^{\mathsf{Bool}}(\iota[Q_0/Q])$, $\mathtt{WorkList} = \{\mathtt{r}\}$
3: while $\mathtt{WorkList} \neq \emptyset$ do
4:     dequeue $n$ from $\mathtt{WorkList}$
5:     $N \leftarrow N \cup \{n\}$
6:     let $(\mathtt{r}, a_1, n_1), (n_1, a_2, n_2), \ldots, (n_{k-1}, a_k, n)$ be the path from $\mathtt{r}$ to $n$
7:     if $\mathsf{Acc}_{\mathcal{A}}(a_1 \ldots a_k)$ is satisfiable then          ▷ counterexample is feasible
8:         get model $(\beta, \nu_1, \ldots, \nu_k)$ of $\mathsf{Acc}_{\mathcal{A}}(\lambda(n))$
9:         return $w = (a_1, \nu_1) \ldots (a_k, \nu_k)$          ▷ $w \in L(\mathcal{A})$ by construction
10:    else          ▷ spurious counterexample
11:        let $\langle \top, I_0, \ldots, I_k, \bot \rangle$ be an interpolant for $\Theta(a_1 \ldots a_k)$
12:        $b \leftarrow \mathtt{false}$
13:        for $i = 0, \ldots, k$ do
14:            if $\Lambda(n_i) \not\models I_i$ then
15:                $\lhd \leftarrow \lhd \setminus \{(m, n_i) \in \lhd \mid m \in N\}$
16:                $\Lambda(n_i) \leftarrow \Lambda(n_i) \wedge I_i$          ▷ strenghten the label of $n_i$
17:                if $\neg b$ then
18:                    $b \leftarrow \textsc{Close}(n_i)$
19:    if $n$ is not covered then
20:        for $a \in \Sigma$ do          ▷ expand $n$
21:            let $s$ be a fresh node and $e = (n, a, s)$ be a new edge
22:            $E \leftarrow E \cup \{e\}$
23:            $\Lambda \leftarrow \Lambda \cup \{(s, \top)\}$
24:            $T \leftarrow T \cup \{(e, \theta_k)\}$
25:            $R \leftarrow R \cup \{(s, \bigcup_{q \in R(n)} \mathsf{FV}^{\mathsf{Bool}}(\Delta(q, a)))\}$
26:            enqueue $s$ into $\mathtt{WorkList}$
27: return true
1: function $\textsc{Close}(x)$ returns Bool
2:     for $y \in N$ such that $\lambda(y) <^* \lambda(x)$ do
3:         if $\Lambda(x) \models \Lambda(y)$ then
4:             $\lhd \leftarrow (\lhd \setminus \{(p, q) \in \lhd \mid q$ is $x$ or a successor of $x\}) \cup \{(x, y)\}$
5:             return true
6:     return false
```

$\langle q_0, x \leq 0 \wedge q_2 \wedge y \geq 0 \rangle$ which strenghtens the label of the endpoint of a from q_1 to $q_1 \wedge x \leq 0 \wedge q_2 \wedge y \geq 0$. The sequence ab is also found to be spurious, which changes the label of its endpoint from $\top$ to $\bot$, and also covers it (depicted with a dashed edge). Since the endpoint of aa is not covered, it is expanded to aaa and aab, in Figure 2 (c). Both sequences aaa and aab are found to be spurious, and the enpoint of aab, whose label has changed from $\top$ to $\bot$, is now covered. In the process, the label of aa has also changed from q_1 to $q_1 \wedge y > x - 1 \wedge q_2$, due to the strenghtening with the interpolant from aab. Finally, the only uncovered node aaa is expanded to $aaaa$ and $aaab$, both found to be spurious, in Figure 3. The refinement of $aaab$ causes the label of aaa to change from q_1 to $q_1 \wedge y > x - 1 \wedge q_2$ and this node is now covered by aa. Since its successors are also covered, there are no uncovered nodes and the procedure returns `true`. □

Theorem 1 *Given an automaton $\mathcal{A}$, such that $L(\mathcal{A}) \neq \emptyset$, Algorithm 1 terminates and returns a word $w \in L(\mathcal{A})$. If Algorithm 1 terminates reporting* `true`*, then $L(\mathcal{A}) = \emptyset$.*

Proof: See [5]. □

We have implemented Algorithm 1 in a prototype tool that uses the Z3 SMT solver[2] for the satisfiability queries and interpolant generation, in the theory of linear integer arithmetic (LIA) combined with booleans. The implementation is available at: `https://github.com/cathiec/AltImpact`.

[2]`https://github.com/Z3Prover/z3`

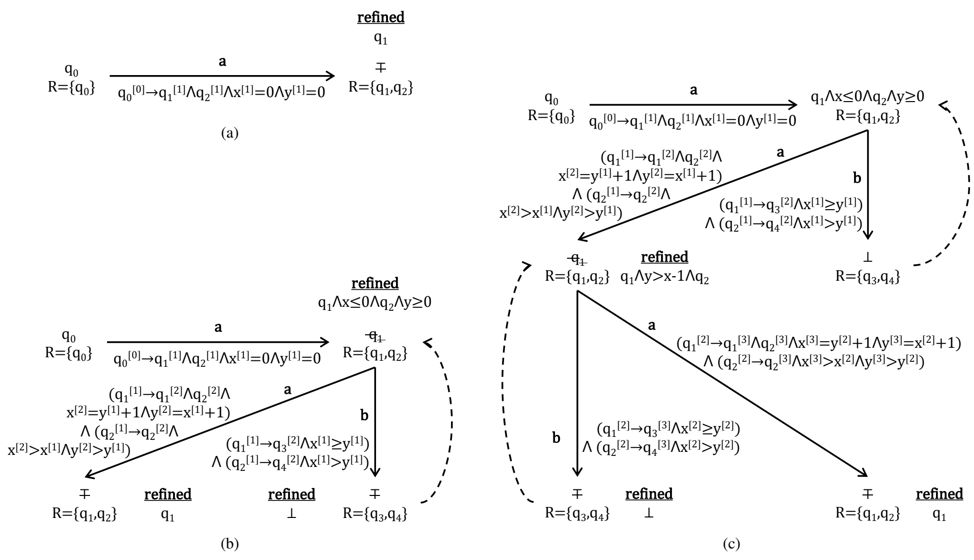
Figure 2: Proving Emptiness of the Automaton from Fig. 1 by Algorithm 1 (1/2)

We compared this algorithm with a previous implementation of a trace inclusion procedure, called Includer[3], that uses on-the-fly determinisation and lazy predicate abstraction with interpolant-based refinement [4] in the LIA theory, without booleans.

References

[1] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification*, LNCS, pages 171–177. Springer, 2011.

[2] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[3] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 337–340. Springer, 2008.

[4] R. Iosif, A. Rogalewicz, and T. Vojnar. *Abstraction Refinement and Antichains for Trace Inclusion of Infinite State Systems*, pages 71–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[5] R. Iosif and X. Xu. The impact of alternation. *CoRR*, abs/1705.05606, 2017.

[6] B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, CAV'2009*, volume 5643 of *LNCS*, pages 661–667, 2009. http://apron.cri.ensmp.fr/library/.

[7] K. L. McMillan. Lazy abstraction with interpolants. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[8] R. Piskac, P. Suter, and V. Kuncak. On decision procedures for ordered collections. Technical report, 2010.

[3] http://www.fit.vutbr.cz/research/groups/verifit/tools/includer/

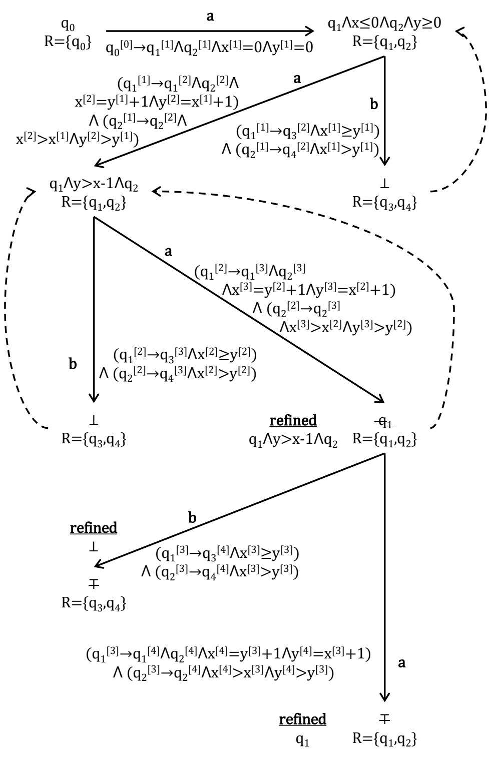
q_0
$R=\{q_0\}$
a
$q_0^{[0]} \rightarrow q_1^{[1]} \wedge q_2^{[1]} \wedge x^{[1]}=0 \wedge y^{[1]}=0$
$q_1 \wedge x \leq 0 \wedge q_2 \wedge y \geq 0$
$R=\{q_1,q_2\}$
a
$(q_1^{[1]} \rightarrow q_1^{[2]} \wedge q_2^{[2]} \wedge x^{[2]}=y^{[1]}+1 \wedge y^{[2]}=x^{[1]}+1) \wedge (q_2^{[1]} \rightarrow q_2^{[2]} \wedge x^{[2]}>x^{[1]} \wedge y^{[2]}>y^{[1]})$
b
$(q_1^{[1]} \rightarrow q_3^{[2]} \wedge x^{[1]} \geq y^{[1]}) \wedge (q_2^{[1]} \rightarrow q_4^{[2]} \wedge x^{[1]}>y^{[1]})$
$\perp$
$R=\{q_3,q_4\}$
$q_1 \wedge y>x-1 \wedge q_2$
$R=\{q_1,q_2\}$
a
$(q_1^{[2]} \rightarrow q_1^{[3]} \wedge q_2^{[3]} \wedge x^{[3]}=y^{[2]}+1 \wedge y^{[3]}=x^{[2]}+1) \wedge (q_2^{[2]} \rightarrow q_2^{[3]} \wedge x^{[3]}>x^{[2]} \wedge y^{[3]}>y^{[2]})$
b
$(q_1^{[2]} \rightarrow q_3^{[3]} \wedge x^{[2]} \geq y^{[2]}) \wedge (q_2^{[2]} \rightarrow q_4^{[3]} \wedge x^{[2]}>y^{[2]})$
$\perp$
$R=\{q_3,q_4\}$
refined
$q_1 \wedge y>x-1 \wedge q_2$
~~q_1~~
$R=\{q_1,q_2\}$
b
$(q_1^{[3]} \rightarrow q_3^{[4]} \wedge x^{[3]} \geq y^{[3]}) \wedge (q_2^{[3]} \rightarrow q_4^{[4]} \wedge x^{[3]}>y^{[3]})$
refined
$\perp$
~~$\top$~~
$R=\{q_3,q_4\}$
a
$(q_1^{[3]} \rightarrow q_1^{[4]} \wedge q_2^{[4]} \wedge x^{[4]}=y^{[3]}+1 \wedge y^{[4]}=x^{[3]}+1) \wedge (q_2^{[3]} \rightarrow q_2^{[4]} \wedge x^{[4]}>x^{[3]} \wedge y^{[4]}>y^{[3]})$
refined
q_1
~~$\top$~~
$R=\{q_1,q_2\}$

Figure 3: Proving Emptiness of the Automaton from Fig. 1 by Algorithm 1 (2/2)