Comparing EventB, log and Why3 Models of Sparse Sets
Maximiliano Cristiá, Catherine Dubois

To cite this version:
Maximiliano Cristiá, Catherine Dubois. Comparing EventB, log and Why3 Models of Sparse Sets. 35es Journées Francophones des Langages Applicatifs (JFLA 2024), Jan 2024, Saint-Jacut-de-la-Mer, France. hal-04407130

HAL Id: hal-04407130
https://hal.science/hal-04407130
Submitted on 22 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Comparing EventB, \{log\} and Why3
Models of Sparse Sets

Maximiliano Cristiá\(^1\) and Catherine Dubois\(^2\)

\(^1\)Universidad Nacional de Rosario and CIFASIS, Rosario, Argentina
\(^2\)ENSIIE, Inria, Université Paris-Saclay, LMF, France

Many representations for sets are available in programming languages libraries. This paper focuses on sparse sets and two of their operations used in some constraint solvers for representing integer variable domains, which are finite sets of values, as an alternative to range sequences. We formalize this data structure and two of its operations and prove their correctness, in three deductive formal verification tools, EventB, \{log\} and Why3. Furthermore, we draw some comparisons regarding specifications and proofs.

1 Introduction

Sets are widely used in programs. They are sometimes first-class objects of programming languages, e.g. SETL [23] or \{log\} [13], but more frequently they are data structures provided in libraries. Many different representations are available, depending on the targeted set operations. In this paper, we deal with sparse sets, introduced by Briggs and Torczon [6], used in different contexts and freely available for different programming languages (Rust, C++ and many others). We focus on their use in constraint solvers as an alternative to range sequences or bit vectors for implementing domains of integer variables [19] which are nothing else than mathematical finite sets of integers. With such an implementation, searching and removing an element are constant-time operations. Furthermore sparse sets are cheap to trail and restore, which is a key point when backtracking for finding solutions.

Confidence in constraint solvers using sparse sets can be improved if the algorithms implementing the main operations are formally verified, as it has been done by Ledein and Dubois [20] for the traditional implementation of domains as range sequences. Hence, the main contribution of this paper is a verified implementation of integer variable domains as sparse sets and the main operations used in constraint solvers (i.e. remove and bind) in EventB, \{log\} and WhyML and their associated verification tools. We prove that the implemented operations preserve the invariant properties and we also express and prove properties that can be seen as formal foundations of trailing and restoring. As far as we know, this is the first formally verified implementation of some operations on sparse sets. All specifications and proofs can be found here: https://gitlab.com/cdubois/SparseSets.

We have chosen to use EventB and \{log\} as representatives of set-based formalisms, both providing native high level operators on sets, relations and functions. However, in this family they are quite different: the former has an imperative flavor and offers refinement as a development method where models are specified gradually; the latter is based on the constraint logic programming (CLP) paradigm. Why3 is representative of deductive program verification tools manipulating contracts.
A second contribution of this paper is a comparison of these three formalizations with respect to aspects such as expressiveness, specification analysis and automated proof.

2 Sparse sets

We deal here with sets as subsets of natural numbers up to \(N-1\), where \(N\) is any non-zero natural number. A sparse set \(S\) is represented by two arrays of length \(N\) called \(\text{mapD}\) and \(\text{domD}\), and a natural number \(\text{sizeD}\). The array \(\text{mapD}\) maps any value \(v \in [0, N-1]\) to its index \(\text{ind}_v\) in \(\text{domD}\), the value indexed by \(\text{ind}_v\) in \(\text{domD}\) is \(v\). The main idea that brings efficiency when removing an element or testing membership is to split \(\text{domD}\) into two sub-arrays, \(\text{domD}[0, \text{sizeD} - 1]\) and \(\text{domD}[\text{sizeD}, N-1]\), containing resp. the elements of \(S\) and the elements of \([0, N-1]\) not in \(S\). Then, if \(S\) is empty (resp. the full set), \(\text{sizeD}\) is equal to 0 (resp. \(N\)). Fig. 1, inspired from a figure in [19], illustrates this representation.

Checking if an element \(i\) belongs to the sparse set \(S\) simply consists in the evaluation of the expression \(\text{mapD}[i] < \text{sizeD}\). Removing an element from the set consists in moving this element to \(\text{domD}[\text{sizeD}, N-1]\) (with 2 swaps in \(\text{mapD}\) and \(\text{domD}\) and decreasing \(\text{sizeD}\)). Binding \(S\) to a single element of the set \(S\) follows the same idea: moving this element at the first place in \(\text{domD}\) and assigning the value 1 to \(\text{sizeD}\).

In our formalization, we only deal with two operations consisting in removing an element in a sparse set and binding a sparse set to a singleton set since these two operations are fundamental when solving constraints. Removing is necessary when domains are pruned. Binding a sparse set to a singleton set is done when the solver assigns variables. The solver never inserts values in a domain. Solvers may also need to walk through all the elements of a variable domain, exploring \(\text{domD}[0..\text{sizeD} - 1]\). This is outside the scope of this work but it presents no particular difficulty.

Many constraint solvers use a data structure called \textit{trail} to store undo information (such as domains) when backtracking on possible solutions. When sparse sets are used, only \(\text{sizeD}\) needs to be kept in the trail. Domains can, then, be restored in constant time by setting the \(\text{sizeD}\) variable back to its previous value [19].

Quoting Le Clément de Saint-Marcq et al. [19], there are three key predicates that should be invariants of any sparse set implementation:

\begin{itemize}
  \item \(D = \{ \text{domD}[i] | 0 \leq i \leq \text{sizeD} \}\) \hfill (P_1)
  \item \(\text{mapD}[v] = i \Leftrightarrow \text{domD}[i] = v\), for all \(i\) and \(v\) \hfill (P_2)
  \item \(\text{domD}[\text{sizeD}..N-1]\) remains unchanged. \hfill (P_3)
\end{itemize}

These properties have been proved to hold in the three formalizations analyzed in this paper.

3 EventB formal development

In this section we succinctly introduce the EventB formal specification language and in more details the EventB models for sparse sets.
Formal Models of Sparse Sets

MACHINE Domain
VARIABLES D
INVARIANTS inv1: D \subseteq 0..N – 1
Initialisation begin act1: D := 0..N – 1 end
Event remove \equiv
  any v where grd1: v \in D
  then act1: D := D \setminus \{v\} end

Figure 2. EventB abstract specification, the Domain machine

3.1 EventB and Rodin

EventB [4] is a deductive formal method based on set theory and first order logic allowing users to design correct-by-construction systems. It relies on a state-based modeling language in which a model, called a machine, is made of a state characterized by variables and a collection of events describing state changes. The state consists of variables constrained by invariants. Proof obligations are generated to verify the preservation of invariants by events. A machine may use a context which introduces abstract sets, constants, axioms or theorems. A formal design in EventB starts with an abstract machine which is usually refined several times. Proof obligations are generated to verify the correctness of a refinement step.

An event may have parameters. When its guards are satisfied, its actions, if any, are executed, updating state variables. Actions may be -multiple- deterministic assignments, x, y := e, f, or -multiple- nondeterministic ones, x, y := \{BAP(x, x', y, y')\} where BAP is called a Before-After Predicate relating current (x, y) and next (x', y') values of state variables x and y. In the latter case, x and y are assigned arbitrary values satisfying the BAP predicate. When using such a non-deterministic form of assignment, a feasibility proof obligation (FIS) is generated in order to check that there exist values for x' and y' such that \{BAP(x, x', y, y')\} holds. In the latter case, x and y are assigned arbitrary values satisfying the BAP predicate. A dedicated proof obligation called simulation (SIM) is automatically generated.

In the following, we use Rodin, an Eclipse based IDE for EventB project management, model edition, refinement and proof, automatic proof obligations generation, model animation and code generation. Rodin supports automatic and interactive provers [16]. In this work we used the standard provers (AtelierB provers) and also the SMT solvers VeriT [3], CVC3 [1] and CVC4 [2]. More details about EventB and Rodin can be found in [4] and [5].

3.2 EventB formalization

The formalization is made of six components, i.e. two contexts, a machine and three refinements. Context Ctx1 introduces the bound N as a non-zero natural number and context Ctx2 extends the latter with helper theorems. The high level machine gives the abstract specification. This model contains a state composed of a finite set D, constrained to be a subset of the (integer) range 0..N – 1, and two events, to remove an element from D or set D as a singleton set (see Fig. 2).

The first refinement (see Fig. 3) introduces the representation of the domain as a sparse set, i.e. two arrays mapD and domD modeled as total functions (inv1 and inv2) and also the variable sizeD which is a natural number in the range 0..N (inv3). Invariants inv4 and inv5 constrain mapD and domD to be inverse functions of each other (property P2 of Sect. 2). The gluing invariant inv6 relates the states between the concrete and former abstract machines1. So the set domD[0..sizeD – 1] containing the elements of the subarray from 0 to

1In a refinement relationship, the machine which is refined is called the abstract machine whereas the refinement is called the concrete machine.
Formal Models of Sparse Sets
Maximiliano Cristiá and Catherine Dubois

MACHINE SparseSets ref1
REFINES Domain
SEES Ctx1

VARIABLES domD mapD sizeD

ININVARIANTS
  inv1: domD ∈ 0..N − 1 → 0..N − 1
  inv2: mapD ∈ 0..N − 1 → 0..N − 1
  inv3: sizeD ∈ 0..N
  inv4: domD = id0..N − 1
  inv5: mapD : domD = id0..N − 1
  inv6: domD[0..sizeD − 1] = D
  inv7: ( theorem ) ∀ x, v : x ∈ 0..N − 1 ∧ v ∈ 0..N − 1 ⇒ (mapD(v) = x ⇔ domD(x) = v)
  inv8: ( theorem ) domD ∈ 0..N − 1 → 0..N − 1

Initialisation begin
  act1: mapD, domD := id0..N − 1, id0..N − 1
  act2: sizeD := N end

Event remove ≜ refines remove
  any v where v ∈ 0..N − 1 ∧ 0 < sizeD ∧ mapD(v) < sizeD
  then
    mapD, domD, sizeD := domD′ ∈ 0..N − 1 → 0..N − 1 ∧ mapD′ ∈ 0..N − 1 → 0..N − 1
    ∧ domD′ := id0..N − 1 ∧ mapD′ := id0..N − 1
    ∧ domD′[0..sizeD′ − 1] = domD[0..sizeD − 1] \ { v } ∧ sizeD′ < sizeD
    ∧ (sizeD . N − 1) \ { domD′ = (sizeD . N − 1) \ { DomD } } end

Figure 3. EventB first refinement (excerpt)

Event remove ≜ refines remove
  any v where v ∈ 0..N − 1 ∧ 0 < sizeD ∧ mapD(v) < sizeD
  then
    act1: domD := domD \ { mapD(v) } → domD(domD − 1), sizeD − 1 → v
    act2: mapD := mapD \ { v → sizeD − 1, domD(sizeD − 1) → mapD(v) }
    act3: sizeD := sizeD − 1

Figure 4. EventB Event remove in the second refinement

sizeD − 1 is exactly the set D. This is exactly property P1 of Sect. 2.

Theorems inv7 and inv8 are introduced to ease interactive proofs, they are proved as consequences of the previous formulas (inv1 to inv6). inv7 follows directly from a theorem of Ctx1 whose statement is inv7 where domD and mapD are universally quantified. Theorem inv8 states that domD is an injective function.

Variables mapD and domD are both initialized to the identity function on 0..N − 1 (denoted id0..N − 1) and sizeD to N. Events of the initial machine are refined by non deterministic events. Thus remove assigns the three state variables with any values that satisfy invariants and also such that sizeD strictly decreases and removed elements in domD are kept at the same place (properties in bold font). The \ { } operator computes the domain restriction of a function or relation. Event bind, omitted in Fig. 3 for lack of space, follows the same pattern. The only reason to have introduced this intermediate model SparseSets_ref1 is to express the properties written in bold font, one of them being the property P1 of Sect. 2. In fact, because they relate two states, they cannot be expressed as invariants.

The second refinement has the same state than the previous one (see Fig. 4). Its events implement the operations and are a straightforward translation of the algorithms in [19].

To discharge the FIS proof obligations of SparseSets_ref1, we can use the values of domD, mapD and sizeD specified in SparseSets_ref2 as witnesses. The SIM proof obligations of SparseSets_ref2 require to prove that the latter values again satisfy the BAP predicate used in SparseSets_ref1. In order not to do these interactive proofs twice, we generalize them and prove them as theorems of the context. In this way, to provide a proof of the FIS and
SIM proof obligations, we only have to instantiate these theorems.

4 \{log\} formal development

In this section we briefly present the \{log\} tool and how we used it to encode sparse sets.

4.1 \{log\}

\{log\} is at the same time a CLP language and satisfiability solver where sets and binary relations are first-class citizens [21, 17, 8]. The tool implements several decision procedures for expressive fragments of set theory and set relation algebra including cardinality constraints [15], restricted universal quantifiers [14], set-builder notation [10] and integer intervals [12]. Case studies developed with \{log\} can be consulted in [9, 11, 7].

\{log\} code enjoys the formula-program duality meaning that \{log\} code can behave as both a formula and a program. When seen as a formula, it can be used as a specification on which verification conditions can be (sometimes automatically) proved. When seen as a program, it can be executed. Thus \{log\} code is sometimes called forogram—a portmanteau word resulting from combining formula with program.

In the following formalization, we use the (still under development) state machine specification language (SMLS) defined on top of \{log\}. SMLS provides declarations very close to those of EventB to declare state variables (variables), operations (operation) and invariants (invariant). The latter is used to automatically generate verification conditions (VC) (proof obligations) on state machines. Users can use \{log\} itself to automatically prove or disprove these VCs [22]. Unlike EventB, SMLS does not support the notion of refinement.

4.2 \{log\} formalization

The \{log\} formalization presented in this paper uses a combination of CLP and set-based, state-based specifications. While CLP is at the core of \{log\}, set-based, state-based specifications can be easily written by means of SMLS. Fig. 5 and 6 list representative parts of the \{log\} forgram in which we use the same identifiers as for the EventB models as much as possible, within the syntactic constraints of \{log\}.

The \{log\} forgram is mainly a state machine described with SMLS modifying 4 state variables (DomD, MapD, SizeD, D) by 2 operations, remove and bind (see Fig. 5). The two arrays are modeled by total functions and their typing constraints become invariant properties as in EventB (split here in small predicates to increase the chances of automated proofs). Property P2 of Sect. 2 is also an invariant of this state machine (inv4 and inv5, the latter, omitted in the figure, is the symmetric of the former). Parameter I is used to compute the identity relation on the integer interval [0, N - 1] as shown in axiom axm2, which in turn is used in invariant inv4. \{log\} inherits many of Prolog’s features. In particular, integer expressions are evaluated by means of the is predicate. Along the same lines, all set operators are implemented in \{log\} as constraints. For example, id(A,R) is true when R is the identity relation on set A. The term int(0,M) corresponds to the integer interval [0, M]. Assertion inv7 is introduced to help the solver\(^2\), it can be deduced from previous invariants (as in Fig. 3). Therefore, we introduce it as a simple predicate but then we declare a theorem (inv7_th) whose conclusion is inv7 but in a negated form because \{log\} is a satisfiability solver. Later, \{log\} will include inv7_th as a proof obligation and will attempt to discharge it. In inv7, the foreach constraint implements the notion of restricted universal quantifier (RUQ). That is, for some \{log\} formula \(\phi\) and set \(A\), foreach(X in A, \(\phi(X)\)) corresponds to \(\forall X. (X \in A \Rightarrow \phi(X))\) where \(A\) can be a set or a binary relation. In the latter case, the quantified expression can be an ordered pair, as is the case of inv7 and inv6 (in Fig. 6). \{log\} also offers the exists constraint implementing the notion of

\(^2\)Without it, some proofs are not automatic.
The remove operation is encoded as a \{log\} predicate. State variables are included as explicit arguments since in \{log\} there is no global state. Next-state variables are denoted by adding an underscore character to the base name (SizeD_). Set unification is used to implement function application. For instance, DomD = \{[S,Y2],[Y1,Y5] / DomD1\} is equivalent to: \[\exists y_2, y_5, \text{DomD}_1\{\text{SizeD} - 1, y_2\} \cup \text{DomD}_1\}, where \(y_1 = \text{MapD}(v)\) (due to the previous set unification). Non-membership constraints following the equality constraint prevent \{log\} from generating repeated solutions. Hence, when remove is called with a set term in its fourth argument, this term is unified with \{[S,Y2],[Y1,Y5] / DomD1\}. If the unification succeeds, then the images of S and Y1 are available.

**Figure 5.** Some representative axioms, invariants and operations of the \{log\} forgram.

The state machine is complemented with some user-defined proof obligations (see Fig. 6) which are introduced as theorems to ensure that the \{log\} forgram verifies properties \(P_1\) (inv6 in the forgram) and \(P_3\). Precisely theorem remove_pi_inv6 states that if inv6 holds and remove and its abstract version\(^3\) (not shown in the paper) are executed, then inv6 holds in the next state. Likewise, theorem remove_b2 ensures that if remove is executed and the functional image\(^4\) of the interval int(SizeD,N-1) through DomD_ is FI, then it must coincide with the functional image of the same interval but through DomD.

The VCs generated by \{log\} include satisfiability of the conjunction of all axioms, satisfiability of each operation and invariance lemmas for each and every operation and invariant. For invariance lemmas, \{log\} includes a minimum set of hypotheses in order to have to solve a simpler goal, reducing the possibilities of a complexity explosion. Hypotheses can be manually added and the proof run again. This process can be iterated until all proofs are done—or the complexity explosion cannot be avoided. The command findh helps the user to find missing hypotheses. \{log\} discharges all the VCs for the complete forgram.

\(^3\)remove and its abstract version can be distinguished by their arities.

\(^4\)fimg is a user-defined \{log\} predicate computing the relational image through a function.
Figure 6. User-defined proof obligations

5 Why3 formal development

In this section we briefly introduce the Why3 platform and describe with some details our specification of sparse sets.

5.1 WhyML and Why3

Why3 [18] is a platform for deductive program verification providing a language for specification and programming, called WhyML. It relies on external automated and interactive theorem provers, to discharge VCs. Here we used the SMT provers CVC4 and Z3. Proof tactics are also provided, making Why3 a proof environment close to the one of Rodin for interactive proofs. Why3 supports modular verification.

WhyML allows the user to write functional or imperative programs featuring polymorphism, algebraic data types, pattern-matching, exceptions, references, arrays, etc. These programs can be annotated by contracts and assertions and thus verified. User-defined types with invariants can be introduced, the invariants are verified at the function call boundaries. Furthermore to prevent logical inconsistencies, Why3 generates a verification condition to show the existence of at least one value satisfying the invariant. To help the verification, a witness is explicitly given by the user (by clause in Fig. 7). The old operator can be used inside post-conditions to refer to the value of a term at the call program point. Programs may also contain ghost variables and ghost code to facilitate specification and verification. From verified WhyML programs, correct-by-construction OCaml programs (and recently C programs) can be automatically extracted.

5.2 Why3 formalization

We first define a record type, sparse, whose mutable fields are a record of type sp_data containing the computational elements of a sparse set representation and a ghost finite set of integer numbers which is the abstract model of the data structure. The type invariant of sparse relates the abstract model with the concrete representation as in Property $P_1$ of Sect. 2. It is used to enforce consistency between them. Invariants enforcing consistency between the two arrays mapD and domD and the bound sizeD are attached to the sp_data type: length of the arrays is n, contents are belonging to the integer range 0..n − 1 and the two arrays are inverse of each other (Property $P_2$), sized is in 0..n. These type definitions and related predicates are shown in Fig. 7.

Our formalization contains three functions, swap_sp_data, remove_sparse (see Fig. 8) and bind_sparse (omitted here), which update their arguments. They are the straightforward translation of the algorithms in [19], except for the supplementary ghost code (last
predicate ran (a: array int) (n: int) = 
0 <= n && a.length = n && forall i. 0<=i<n -> 0<=a[i]< n

type sp_data = {n: int; mutable domD, mapD : array int; mutable sizeD: int; }

invariant {ran domD n && ran mapD n && 0 <= sizeD <= n &&
forall v, i. (0<=i<n && 0<=v<n) -> (domD[i]=v <-> mapD[v]=i)} by ...

type sparse = {mutable data: sp_data; mutable ghost setD: fset int;}

invariant {subset setD (interval 0 data.n) &&
forall x: int.(exists i:int. 0 <= i < data.sizeD && x = data.domD[i]) <-> mem x setD} by ...

Figure 7. WhyML types for sparse sets

predicate same_end (a : array int) (b : array int) (s : int) (n : int) =
forall i. s <= i < n -> a[i] = b[i]

let swap_sp_data (a : sp_data) (i : int) (j : int)
requires {0<=i<a.n && 0<=j<a.n}
ensures {exchange (old a.domD) a.domD i j}
ensures {exchange (old a.mapD) a.mapD a.domD[i] a.domD[j]} =
swap a.domD i j; a.mapD[a.domD[i]] <- i; a.mapD[a.domD[j]] <- j;

let remove_sparse (v : int) (a : sparse)
requires {0<=v<a.data.n && a.data.mapD[v] < a.data.sizeD && a.data.sizeD > 0}
ensures {old a.data.sizeD > a.data.sizeD}
ensures {same_end a.data.domD (old a.data.domD) (old a.data.sizeD) a.data.n} =
swap_sp_data a.data a.data.mapD[v] (a.data.sizeD - 1);

a.data.sizeD <- a.data.sizeD - 1;
a.setD <- remove v a.setD

Figure 8. WhyML functions for sparse sets

statement in remove_sparse which updates the abstract model contained in a.setD. Function swap_sparse_data is a helper function. The contract of swap_sparse_data makes explicit the modifications of both arrays a.mapD and a.domD, using the exchange predicate defined in the library. VCs for this function concern the conformance of the code to the two post-conditions (trivial as it is ensured by swap) and also the preservation of the invariant attached to the sparse_data type—i.e. mainly that a.mapD and a.domD after swapping elements remain inverse from each other. Both remove_sparse and bind_sparse act not only on the two arrays and the bound but also on the ghost part, i.e. the corresponding mathematical set a.setD. Thus VCs here not only concern the structural invariants related to mapD, domD and sizeD but also the ones deriving from the use of the sparse type, proving the link between the abstract logical view (using finite sets) and the computational one implemented through arrays. The property P3 is expressed here as a post-condition.

All proofs are discovered by the automatic provers except for some proof obligations related to the remove function. Nevertheless these interactive proofs remain simple thanks to some Why3 tactics that inject some hints to help external provers to finish the proofs.

6 Comparison and discussion

Clearly, all three formalisms and tools are expressive enough for the problem at hand. They all allow axioms, invariants and operations to be expressed. The EventB specification is probably the most readable. Properties P1 and P2 of Sect. 2 emphasised in [19] are expressed
as invariants in the three formalisms. Property $P_3$ about the removed part of the domain, which must relate two states, is expressed as a post-condition of the operations.

Writing $P_3$ in EventB proved to be complex. In fact, it was necessary to add a somewhat artificial level of refinement for Rodin to be able to generate the desired VCs link. This property can be more easily defined in \{log\} and Why3.

In general, all three tools automatically generate similar VCs. However, in Why3 and EventB, abstract and concrete models can be naturally linked through refinement or ghost code and the tools automatically generate the corresponding VCs. \{log\} still needs work to express how two models are linked in terms of abstraction/refinement relations. All VCs in EventB and Why3 are automatically generated, which is not the case in \{log\}, making the \{log\} version of our verification effort less trustworthy than Why3 and Rodin.

Table 1 summarizes the results of the three verification efforts (for the two operations). The first column gives the number of VCs —numbers in brackets correspond to manually written VCs. The second (resp. third) column contains the number of automatically (resp. interactively) proved VCs.

In EventB, 46 proof obligations were generated (about half of them from the first refinement) of which 34 were automatically proved by the (AtelierB) standard provers and VeriT. For the 12 that were proved interactively, VeriT was very helpful when additional, back-up hypotheses were added. Only two proofs required real human intervention. Using the process described in Sect. 4, \{log\} unloads all 38 VCs in about 7 minutes. Likely the existence of dedicated set-theoretic decision procedures proved crucial, since \{log\} is the only tool that automatically discharges all VCs after a simple hypothesis discovery procedure.

Why3 makes it possible to apply transformations (e.g. split conjunctions) to a proof goal before calling an automatic prover on it. Some of these transformations are very simple, e.g. split conjunctions, and can then be applied systematically and automatically. Most of the VCs generated in our formalization have been proven automatically thanks to the split transformation. Only two of them, both dealing with type invariants, required human interaction to insert some more complex transformations, e.g., a case analysis on indices in mapD (case \(i=a\_data.mapD[v]\)). In the end, 53 VCs were proved—47 by CVC4 and 6 by Z3—in 9 seconds.

7 Conclusion

We formally verified the implementation of two operations on sparse sets using three formal languages and associated tools, focusing on the operations and correctness properties required by a constraint solver when domains of integer variables are implemented with sparse sets. In particular we compared the different statements of the required properties —namely $P_1$, $P_2$ and $P_3$ given in Sect. 2— and their proofs.

As future work, the formal developments can be completed with other operations. A performance evaluation of the extracted code could then be performed. A second line of work is to implement and verify, in Why3 or EventB, a labelling procedure that assigns values to variables, such as those used in constraint solvers, it would be necessary to backtrack on the values of some domains and thus make use of the theorems proved in this paper. Since labelling is native in \{log\} when CLP(FD) [24] is enabled, assignment of values to variables is trivial although less trustworthy than a formally verified algorithm.
References


