Constraint Propagation for Bit-Vectors in Alt-Ergo*

Hichem Rami Ait-El-Hara[1,2], Guillaume Bury[1], Basile Clément[1,*] and Pierre Villemot[1]

[1]OCamlPro SAS, 21 Rue de Chatillon, 75014, Paris, France
[2]Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Abstract

Alt-Ergo is an SMT solver with strong ties to the deductive program verification framework Why3. It is used by industrial program verification tools such as Frama-C, TIS-Analyzer and SPARK. In this paper, we present recent developments to Alt-Ergo and its frontend, Dolmen. We describe a new bit-vector solver currently under development in Alt-Ergo that supports the arithmetic and logic operators from the SMT-LIB "BV" logic. We also describe the changes made to Dolmen to support version 2.7 of the SMT-LIB standard. Alt-Ergo's new bit-vector solver leverages a new constraint propagation architecture, a rewritten interval module, and standard propagators adapted to Alt-Ergo's Shostak-like combination procedure. We provide some preliminary benchmark results, and report on our experience and ongoing challenges.

Keywords

SMT-LIB, Bit-Vectors, Constraint Propagation

1. Introduction

Alt-Ergo is an SMT solver with close historical ties with the Why3 platform for deductive program verification. Historically using an ad-hoc language inspired by Why3's own WhyML, Alt-Ergo now uses the SMT-LIB language through Dolmen, the library powering the tool of the same name. Both are developed at OCamlPro SAS (Alt-Ergo was initially developed by Université Paris-Sud). In addition to powering Alt-Ergo, Dolmen is also used to check compliance of benchmarks submitted to the SMT-LIB benchmark library [1], which are notably used for the SMT-COMP solver competition.

In this paper, we present recent developments in both Alt-Ergo and Dolmen to integrate support for the SMT-LIB theory of bit-vectors ("BV") and the recent version 2.7 of the SMT-LIB standard. We report on these developments and our first experiments with the new BV theory in Alt-Ergo.

The paper is organized as follows. In Section 2, we give an overview of Alt-Ergo's general architecture and some of its key features. In Section 3 we present in more detail our implementation of the bit-vector theory. Section 4 provides experimental evaluation on standard benchmarks and industrial program verification benchmarks. Finally, we discuss future works and conclude our presentation in Section 5.

2. General Architecture

Alt-Ergo historically used a purely functional Tableaux-like solver for boolean constraints, which guided most of the design. The Tableaux solver is still present, but Alt-Ergo now includes a CDCL solver with Tableaux-like extensions that is now used as the default solver. The theory modules are shared between both solvers, and combined by a Shostak-like combination procedure.

SMT 2025: 23rd International Workshop on Satisfiability Modulo Theories, August 10–11, 2025, Glasgow, UK

*This research was supported partly by the Décysif project funded by the Île-de-France region and by the French government in the context of "Plan France 2030".

*Corresponding author.

hichem.ait-el-hara@ocamlpro.com (H. R. Ait-El-Hara); guillaume.bury@ocamlpro.com (G. Bury); basile.clement@ocamlpro.com (B. Clément); pierre.villemot@ocamlpro.com (P. Villemot)

0000-0001-7909-0413 (H. R. Ait-El-Hara); 0000-0001-7116-9338 (G. Bury); 0000-0002-0877-7063 (B. Clément)

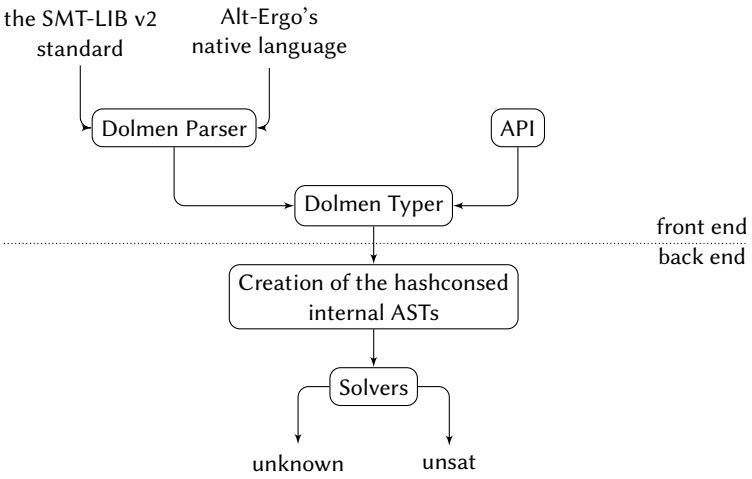

Figure 1: Alt-Ergo's architecture

2.1. Shostak

One of Alt-Ergo's central modules is the one implementing equational reasoning for convex theories. The original algorithm, called CC(X) [2], is heavily inspired by the Shostak decision procedure: it maintains a union-find data structure modulo a theory X, equipped with a solver and a canonizer, from which implied equalities are used for congruence closure. The algorithm has later been extended into another algorithm, AC(X) [3], to also handle associative and commutative user-defined symbols. AC(X) is obtained by augmenting ground AC completion with the canonizer and solver for the theory X.

It is also at this level that case splits coming from theories (rather than from boolean disjunctions in the input formula) are handled.

2.2. Theories

Alt-Ergo supplements the core ("Shostak") algorithm used to decide equality with theory-specific solvers to deal with the non-equational parts of theories. Each solver is responsible for storing the relevant information in its own module. Theory solvers communicate through equalities (that are processed by the CC(X) algorithm to produce substitutions that are then propagated to all theories) and predicates. Each theory is also able to provide the solver with *case-splits* to deal with non-convex theories.

Alt-Ergo provides theory modules for linear arithmetic over integers and rationals, fragments of non-linear arithmetic, polymorphic functional arrays with extensionality, algebraic data types, associative and commutative symbols (built into the AC(X) algorithm), floating-point arithmetic [4], and fixed-size bit-vectors.

Non-linear arithmetic Alt-Ergo uses a combination of the AC(X) framework and interval calculus to handle non-linear integer arithmetic in a built-in way [3]. The AC(X) framework is instantiated with linear integer arithmetic (LIA) to handle linear equalities and the associativity and commutativity properties of non-linear multiplication. NIA axioms are integrated into the interval calculus module of Alt-Ergo in order to propagate the bounds of non-linear terms and to suggest case splits on finite domains.

ADTs Alt-Ergo supports algebraic data types. This is handled through the native combination of Shostak solvers for the theory of records and of enumerated data types. The theory of enumerated data types, and by extension of records, is not convex, so the Shostak solver is not complete. We supplement the Shostak solver with a relational theory that keeps track of possible constructors and then uses the case split mechanism. Within a case split the Shostak solver for records is used.

Floating-point arithmetic Alt-Ergo does not (yet) support the theory of floating-point numbers. Instead, Alt-Ergo supports a theory of a `round` function which returns the rational closest to a real according to a given floating point specification (bits of exponent, bits of mantissa, and rounding mode).

This theory is currently used by tools such as Why3 by wrapping it in an algebraic data type to represent the extra values (infinites, not-a-number, negative zero). We plan on integrating these wrappers in Alt-Ergo itself to provide direct support for the SMT-LIB theory of floats.

2.3. Constraint Propagation

Many theories in Alt-Ergo make use of constraint propagation and store domains on the variables in the problem; however, until recently, each theory solver had its own ad-hoc implementation of domains (intervals for arithmetic, finite sets for enumerated types) and propagators (simplex and additional non-linear propagators for arithmetic, intersection on equality for enumerated types).

In Alt-Ergo 2.6, we introduced a new generic constraint propagation architecture and made it available to theories. Domains are now stored within the same union-find data structure that is used for equational reasoning, while propagators are implemented within each theory's module.

This new architecture allowed us to simplify the existing experimental theory of algebraic data types and merge it with the existing theories for enumerated data types and records while preserving reasoning power. The constraint-based implementation of arithmetic and logic bit-vector operators, described in Section 3, also uses these new capabilities. The linear arithmetic module is the only remaining module that has not been converted to this new architecture yet.

One limitation of the current implementation is that each theory is responsible for performing its own propagations, restricting opportunities for cross-theory propagations: there is currently no mechanism for notifying a theory when a different theory's domain shrinks. Work is ongoing to integrate constraint propagation into Alt-Ergo's main equality and predicate processing loop, opening the path to more flexible scheduling opportunities.

2.4. Dolmen and SMT-LIB 2.7

Alt-Ergo supports the SMT-LIB language as input since Alt-Ergo 2.2 [5]. This support was greatly improved in Alt-Ergo 2.4 with the switch to Dolmen as a frontend, and the development version has recently also added support for the newly released version 2.7 of SMT-LIB, which adds interesting features such as prenex first order polymorphism, higher-order functions (aka maps), and new bit-vector operations. Alt-Ergo has supported polymorphism from the beginning, and there are no current plans to support higher-order functions, so only the new bit-vector operations needed new code to reason about, see Section 3.

That means that most of the changes to support SMT-LIB 2.7 (except for bit-vectors), are in the frontend of Alt-Ergo, where we use the library from the Dolmen project [6, 7, 8] to handle parsing and typechecking of input problems. We will discuss here the changes that were needed in Dolmen to handle the new features of SMT-LIB 2.7.

Polymorphism Alt-Ergo and Dolmen already support polymorphism, through both Alt-Ergo's native language and a polmoyphic extension of SMT-LIB 2.6 [5], so the only work needed was adding the support of sort parameters: while the current code supports locally bound type variables, SMT-LIB 2.7 uses sort/type variables that are declared once with a global scope, but that are then implicitly locally bound in each statement.

Higher-order SMT-LIB 2.7 adds maps, which are encodings of higher-order functions in first-order logic. While this can be easily handled by a regular theory that defines an abstract type (for maps) and the function application operator, there is one part of the specification that makes support for higher-order application a bit tricky: the syntax for regular application is overloaded to also represent application of the higher-order application operator. Correctly handling this requires a fair bit of care

to properly distinguish the regular first-order applications from the higher-order applications using the first-order syntax, which is also made more complex by the presence of polymorphic maps (i.e., polymorphic functions with no term arguments and that return a map).

3. Bit-vectors

We now describe the implementation of the bit-vector theory in Alt-Ergo. The bit-vector theory is implemented by the combination of:

- An existing solver for the core bit-vector theory (concatenations and extractions), extended to support negation – not described in this paper;
- A constraint propagation mechanism (described in this paper) for other operators, providing full support for the SMT-LIB theory of fixed-size bit-vectors.

The implementation of domains and propagators follows Chihani et al. [9] and uses a combination of bit-patterns ("bitlists") and interval domains. This combination is also used by Zeljić et al. [10] in an mcSAT context.

3.1. Core Bit-Vector Theory

The core bit-vector theory with concat and extract is integrated in the Shostak solver, and was recently updated to also support negation. While it is possible to further extend the solver to represent arbitrary bitwise operators as in Cyrluk et al. [11], this approach uses binary decision diagrams to represent the canonical form of bit-vector expressions and is likely to quickly blow up for large expressions. In addition, such an approach would have non-trivial interactions with extensions to the Shostak method used in Alt-Ergo (namely, support for associative-commutative operators [3]).

3.2. Bitlists and Interval Domains

The propagators for bit-vectors rely on two types of domains: bitlists and intervals. These are integrated into the new constraint propagation interface introduced in Alt-Ergo 2.6.

Bitlists Bitlists are domains used to represent the three possible states of bits in a bit-vector: a bit is either set (known to be 1), cleared (known to be 0), or unknown. The concrete representation of bitlists in a computer system is crucial for performance. In Alt-Ergo, we use arbitrary-size integers (provided by the Zarith library, itself using GMP) to represent the domain of x as a pair $\langle {}^1x, {}^ux \rangle$ where ${}^1x$ represents the bits known to be set in x and ${}^ux$ represents the bits that are not known in x (could be either set or cleared).

The use of big integers to represent bitlists allows very efficient implementations for logical bit-vector operations, which we adapt from Chihani et al. [9] and Michel and Van Hentenryck [12].

For addition and subtraction, we use the algorithms from Dougall [13]. These rely on the fact that addition and subtraction have single-bit carries to provide an efficient and precise implementation of the propagators from Michel and Van Hentenryck [12].

The propagator shown in Figure 2 is used for addition (logic operators are extended to bit-vectors by operating independently on each bit). ${}^1x$ (resp. ${}^1y$) is a mask of the bits known to be 1 in x (resp. y), and ${}^ux$ is a mask of the bits whose value in x (resp. y) is unknown. m is the result of the addition when all unknown bits are 0, and M is the result when all unknown bits are 1. ${}^ur$ is a mask of the unknown bits in the result, and ${}^1r$ is a mask of the bits known to be 1 in the result.

The crucial step is in Equation (3): because the carry in a binary addition is at most one bit, overflow is only possible on the bits where m and M disagree, which are added to the unknown bits of x and y.

We do not implement precise propagators for multiplication as that would have a quadratic complexity. Instead, we simply compute the known low bits (effectively providing an implementation of congruence modulo powers of two), and rely on case splits or other integer reasoning to deal with multiplication.

$m = {}^1x + {}^1y$	smallest (unsigned) result	(1)
$M = m + {}^ux + {}^uy$	largest (unsigned) result	(2)
${}^ur = {}^ux \vee {}^uy \vee (m \oplus M)$	overflow is possible where m and M disagree	(3)
${}^1r = m \wedge \neg {}^ur$	bits set in both m and M are set if known	(4)

Figure 2: Propagator for bit-vector addition

Although the SMT-LIB bit-vector theory uses bit-vectors with a specific (parametric) width, our bitlists do not encode the width explicitly, relying on the types to encode this information instead.

Intervals Alt-Ergo already included a module to represent union of intervals, with explanations (justifications) for each of the gaps between intervals. However, this module was found to be hard to extend to properly support bit-vector operations, notably concatenation and extraction. We rewrote the interval module with a parametric implementation providing basic building blocks to manipulate intervals while correctly preserving explanations. Preserving correct explanations is critical for the soundness of the whole solver, and isolating the code that needs to deal with them makes the code easier to reason about and maintain. Those basic building blocks are then used to safely lift high-level operations such as addition, multiplication, and extraction from intervals to unions of intervals. This new interval module is now used for both the bit-vector solver and the arithmetic solver, although we do not yet have direct communication between the intervals stored by both solvers (see Section 2.3).

Note that using union of intervals (rather than just intervals) allows representing precisely intervals for bit-vectors, without loss of precision in case of overflows or underflows. Instead, we simply implement an operator (using the API described below) to implement an `extract` function that returns the n low bits of an interval. For instance, when using 8-bit arithmetic, subtracting 4 from the interval $[0, 241]$ yields $[-4, 237]$; we then call `extract` for the low bits of this interval to obtain the union $[0, 237] \cup [251, 255]$.

The use of unions in the representation is relatively controlled. In particular, we do not create holes in the union from information coming from the (low) bits of a variable. For instance, if x is an 8-bit variable that is known to have its least-significant bit set, its interval is obtained by doubling $[0, 127]$ then adding 1, and so is $[1, 255]$ – not an union of 128 singleton intervals. The cross-domain propagators (see Section 3.3) would be able to create such holes – Chihani et al. [9] found that allowing a single hole per interval was optimal, but we have not implemented this logic in Alt-Ergo.

Ordered Types The interval module is parametric over an *ordered type*. An ordered type extends a type of "finite values" (typically integers or rationals) with a positive and negative infinites, and also provide a successor and a predecessor funtion.

The successor and predecessor functions do not exist on rationals: we use the well-known trick of infinitesimals, i.e., we use pairs $r + n\epsilon$ where r is a rational, n is an integer, and ϵ is an infinitesimal considered smaller than any rational.

This allows the representation of open rational intervals such as $(3.5, 7)$ by closed intervals such as $[3.5 + \epsilon, 7 - \epsilon]$. In practice, infinitesimals are not exposed by the ordered type interface, and the rest of the interval module only makes use of a few documented axioms (e.g., $\mathrm{pred}(\mathrm{succ}(x)) = x$ if x is finite) on the predecessor and successor functions, providing flexibility in the actual implementation of the ordered type interface.

Core Intervals The core of the interval module is a functor taking an ordered type and returning a "core interval" module. We use two representations of intervals: a normalized representation, with type `union` (represented as a sequence of disjoint intervals with an explanation for the gaps) and an

unnormalized one, with type `set` (represented as an arbitrary set of intervals). Intervals are stored in the normalized representation, but the unnormalized representation can be used to implement complex operations.

The core interval API exposes higher order functions to convert from operations on intervals or bounds *without explanations* to core intervals (`union` and `set` types) *with explanations*. Some examples of such functions include:

- The function `map_to_set` computes the image of an union by an arbitrary *isotone* function `f` over intervals, i.e., a function monotone with respect to inclusion (if $I_1 \subseteq I_2$ then $f(I_1) \subseteq f(I_2)$).
- The function `partial_map_inc` computes the image of a *partial* increasing function such as the conversion from reals to integers.
- The function `trisection_map_to_set` is provided to split a union between the values strictly smaller than a central value x, the singleton $\{x\}$ (if within the union) and the values strictly larger than x. This is used to provide piecewise definitions of multiplication, division.

Interval Algebra The rest of the interval module is concerned with lifting arithmetic operations from values to intervals, using the functions provided by the core interval module. Crucially, this *does not need to care about explanations*: implementers only need to make sure to use the appropriate function from the core interval module (depending on e.g., monotony of the implemented function). This principled design makes the code for interval operations clearer and easier to read, as it is not interleaved with concerns about explanations. The design made it possible to implement the new functions related to euclidean division that were required for bit-vector operations with reasonable confidence. This also allowed us to improve the precision of explanations in the implementation of multiplication: the existing implementation used coarser explanations than necessary out of an abundance of caution, but the generic implementation of monotone functions in the core interval module is able to preserve more precise explanations.

3.3. Intra-Domain and Cross-Domain Propagations

Propagators are registered on specific domains of the variables. When the corresponding domain is reduced and changes, all the propagators that are watching this specific domain fire and add added to a queue for further propagation. Depending on the constraint, propagators are registered either only on the bitlist domains (this is the case for bit-wise operators such as `bvor` or `bvand`), only on the interval domains (this is the case for division and remainder), or both bitlist and interval domains.

In addition, special "cross-domain" propagators are used to perform reductions between the bitlist and interval domains. These are taken from Chihani et al. [9], in particular to refine interval bounds from bitlists.

We currently use a simple scheduling strategy, repeated until a fixpoint is reached: first we saturate same-domain propagators separately for each domain, then we saturate cross-domain propagators.

3.4. Case Splits

The bit-vector module is integrated into the case-split mechanism for completeness and model generation. We perform case splits on bit-vectors by selecting a value for the most significant unknown bit from one of the variables in the problem with minimal, non-fully-known, domain size. This is the simplest heuristic that we found to perform reasonably well.

The use of case splits for bit-vectors revealed that the current implementation in Alt-Ergo is too aggressive, and Alt-Ergo can end up spending a lot of time performing unnecessary bit-vector case splits when making a single boolean decision would allow to conclude quickly. Currently, control does not return to the outer CDCL solver until a satisfying bit-vector assignment has been found. We plan on tackling this issue by better integrating the case split mechanism with the outer CDCL solver, and exploring strategies for selecting case splits along the lines of the labelling strategies described in Chihani et al. [9].

3.5. Difference Logic

Alt-Ergo uses a simplex algorithm for linear arithmetic, but does not have a solver for difference logic. Chihani et al. [9] reported making successful use of a difference logic solver to implement a global delta domain.

We implemented an incremental solver for difference logic following Feydy et al. [14]. The solver is currently instantiated for unsigned bit-vector comparisons only; however, we plan on adding support both unsigned and signed bit-vector comparisons and to detect when conversions between signed and unsigned comparisons are possible based on intervals.

The difference logic solver is still experimental and has not been integrated into the main development branch of Alt-Ergo yet. It has not been used for the benchmarks in the experimental evaluation section.

3.6. Conversions Into and From Integers

In order to leverage the simplex algorithm for integer variables, and to better handle conversions between bit-vectors and integers (either using the non-standard `bv2nat` and `int2bv` functions, or the new conversion operators from the SMT-LIB standard version 2.7), we maintain two maps between canonical representatives of bit-vector and integer variables.

Note that theory solvers in Alt-Ergo are mostly independent; hence, we alternate between runs of the constraint propagation described in this paper and runs of the simplex algorithm for pure integer problems. We plan on later improving the integration of the simplex into the constraint propagation mechanism.

This conversion is currently only performed for bit-vector expressions that appear within a `bv2nat` constructor, or within a (bit-vector) comparison such as `bvule`, `bvult`, etc. We are interested in investigating whether performing this conversion for all bit-vector terms, as in int-blasting [15] approaches, could be beneficial.

Map from bit-vectors to integers For a bit-vector variable x, we maintain a map from all the right-shifts of x (defined below) to a corresponding integer variable. If x has type `(_ BitVec w)`, i.e., a bit-vector of width w, we create an integer constant x_s^I to represent the integer value of the extraction of bits $s, s+1, ..., w-1$ of x (using a little-endian convention, so bit $w-1$ is the most significant bit). If we denote $\mathrm{bv2nat}(x)$ the value of a bit-vector x as an integer (as defined in the SMT-LIB specification), the value $\mathrm{bv2nat}(x_s^I)$ is also equal to $\lfloor \mathrm{bv2nat}(x)/2^s \rfloor$, i.e., the right arithmetic shift of $\mathrm{bv2nat}(x)$ by s. Appropriate inequalities to encode this (floored) division are also added to the simplex.

For an extraction $x_{[i,j]}$ of bits $i, i+1, ..., j-1$, we then compute $\mathrm{bv2nat}(x_{[i,j]})$ as $x_i^I - 2^{j-i} x_j^I$. Concatenations are similarly represented by multiplying by the appropriate power of 2.

Using right-shift in this way allows to limit the number of integer variables created to represent extractions to be linear (rather than quadratic) in the bit-vector width.

This map is updated when the canonical representative of a bit-vector equivalence class changes, propagating relevant information on the integer variables for consumption by the simplex.

Map from integers to bit-vectors Integer variables are represented in Alt-Ergo's Shostak solver by a polynomial. We maintain a similar map from polynomials to the corresponding bit-vector variable, which may have been introduced to represent a right-shift.

When the canonical representative of an integer variable changes, updating the map allows to propagate new equalities learned by the simplex (in the integers) to corresponding equalities in the bit-vectors, which can cause further propagations on bit-vector domains.

Conversion of bit-vector arithmetic operators Furthermore, we also convert arithmetic expressions on bit-vectors (in the same context as previously, either in `bv2nat` operations or bit-vector comparisons) into arithmetic expressions on the corresponding integer variables. Unlike int-blasting approaches, we *only* convert arithmetic operation, not bitwise operations.

Table 1
Comparison on the *QF_BV* benchmarks.

solver	unsat	# unique	wtime unsat
Alt-Ergo + BV	3537 (34%)	1	3517s
Bitwuzla	6570 (63%)	226	792s
COLIBRI	4064 (39%)	2	4064s
CVC5	5947 (57%)	0	426s
Yices	5352 (51%)	6	266s
Z3	6022 (58%)	12	248s

For a bit-vector polynomial (using e.g., `bvadd`, `bvmul`, etc.) we keep a canonical version with coefficients computed modulo 2^n (where n is the bit-width). For each distinct bit-vector with the same canonical polynomial we also have a constant k defined so that $\mathrm{bv2nat}(x) = p + 2^n k$ (where p is the canonical polynomial of x and n is the bit-width of x). We also record appropriate bounds for $\mathrm{bv2nat}(x)$.

When there is a relation between bit-vectors (e.g., an equality or a comparison), we replicate the relation on the integer variables using the stored polynomial and offset, which is then picked up by the simplex.

Note that this design is quite different from that in Chihani et al. [9], where only a single type (integers) is used to represent bit-vectors. In Alt-Ergo, we currently use two types because the use of a Shostak-like combination procedure requires having a canonical form per term, but we want to be able to have both integer (polynomial) and bit-vector (concatenation) canonical forms. This is important so that Alt-Ergo does not lose existing reasoning capabilities around bit-vectors, but requires more work for conversion between the two representations.

4. Experimental Evaluation

In this section, we evaluate Alt-Ergo's bit-vector capabilities on a set of benchmarks coming from program verification, on a subset of the SMT-LIB QF_BV benchmarks, and on some manually crafted examples. Note that on these benchmarks our implementation still suffers from the "agressive case splits" problem described in Section 3.4 and locks us into a naive strategy. We expect further improvements from re-architecting around the issue.

All the benchmarks below use a time limit of 20 seconds. We only report *unsat* results.

QF_BV subset We first compare the new solver against other solvers on a random subset of 10498 problems from the QF_BV category of the SMT-LIB benchmarks library [1] (the 2022 version of the benchmarks has been used). This subset has been generated using the provided `selection.py`[1] script with a seed of 42, a ratio of 25% and a minimum of 300 benchmarks.

The QF_BV category only contains quantifier-free bit-vector problems. It is a good proxy to evaluate the strength of the ground bit-vector solver in isolation, but does not necessarily presume of the performance of real-world usage for more complex program verification tasks typically involving quantifiers and multiple theories.

We compare Alt-Ergo with the bit-vector solver against Z3 4.14.1 [16], CVC5 1.2.1 [17], Bitwuzla 0.7.0 [18], Yices 2.6.5 [19] and COLIBRI 2025.02 [20] (COLIBRI implements a similar propagators-based approach) in Table 1. The *unique* column indicates the number of problems for which a solver is the only one that returned *unsat*. In *wtime unsat* column, we focused on the subset of 3,208 problems solved as unsatisfiable by all solvers in Table 1. The *wtime unsat* is the wall time (in seconds) spent by the solver over this specific subset.

[1]The script is available in the SMT-COMP repository:
https://github.com/SMT-COMP/smt-comp/blob/master/tools/selection/selection.py

Table 2
Comparing solvers on *J3* benchmarks.

solver	unsat	% solved	# unique	wtime unsat
Alt-Ergo (ax)	4940	55		
Alt-Ergo + BV	6262	69%	84	2406s
CVC5	7464	83%	1224	1374s
Z3	5474	61%	9	766s

The new solver's performance is slightly below that of COLIBRI, which is expected – while Alt-Ergo's new solver is inspired by the one in COLIBRI, some architectural constraints in Alt-Ergo prevents us from implementing some crucial reasoning steps. Bitwuzla dominates the competition, especially in terms of unique problems solved, but does not support the same range of theories as the other solvers.

Benchmarks from program verification Since Alt-Ergo is frequently used within the Why3 toolchain, it is relevant to compare the performance of the new solver against other solvers using this toolchain. We now focus on an internal benchmark set of 9,038 industrial problems provided by one of our partners in the DéCySif project. These problems have been generated from verification conditions of C programs using the J3 tool. They are translated into SMT-LIB by Why3 [21] using solver-specific drivers, so that all solvers do not get the exact same SMT-LIB input.

Unlike the QF_BV category of SMT-LIB benchmarks that only compares the ground bit-vector solvers, this benchmark set provides a more contextual comparison. The problems involve bit-vectors and other SMT features such as uninterpreted functions, uninterpreted sorts, floats and integers and are more representative of real-world program verification workflows.

For Alt-Ergo, two different drivers are used: a legacy driver using a Why3-provided axiomatization of bit-vectors, and a new driver using the SMT-LIB BV primitives developed in this work (shown as "Alt-Ergo + BV" in the tables).

We compare the performance of solvers on the J3 dataset in Table 2. The "Alt-Ergo (ax)" solver is the development version of Alt-Ergo *without* the new bit-vector solver, and instead uses the Why3 axiomatization. The "Alt-Ergo + BV" solver is the new bit-vector solver described in this paper. The table shows that the new bit-vector solver provides a significant improvement to Alt-Ergo's reasoning abilities on this dataset. We also compare the performance of Z3 4.14.1 and CVC5 1.2.1 on this benchmark set. We did not take Bitwuzla and Yices on because they do not support all the theories used in these benchmarks: Bitwuzla does not support uninterpreted sorts and Yices does not support FP theory. We are not retaining results for COLIBRI because it encountered errors on most of these files. The *unique* column provides the number of benchmarks that are only solved by a specific solver. In *wtime unsat* column, we focused on the subset of 5,335 problems solved as unsatisfiable by all solvers in Table 2. The *wtime unsat* is the wall time (in seconds) spent by the solver over this specific subset.

The new solver solves more problems than Z3. It also solves less problems than CVC5 overall, but is still able to provide a reasonable contribution in terms of unique results, providing complementarity.

Manually crafted examples Finally, we examine the behavior of different solvers on some simple problems with bit-vector-to-int conversions. We compare Alt-Ergo (with the bit-vector solver) to Z3 4.13.4 and CVC5. Note that we use the non-standard `bv2nat` rather than the new `ubv_to_int` function from SMT-LIB 2.7 as `ubv_to_int` is not supported by all solvers.

The first two examples, shown in Figure 3, state a simple property relating the conversion to integers of the low and high bits of a bit-vector of width 64. These examples illustrate a strength of integration with the existing core bit-vector solver and the Shostak combination procedure in Alt-Ergo: while Z3 is able to quickly prove the first example with `concat`, it fails to do so when using the logically equivalent formulation using `extract`. On the other hand, the Shostak-based solver in Alt-Ergo effectively normalizes the second example to the first, and Alt-Ergo is able to prove both.

```
(declare-const x (_ BitVec 64))
(declare-const hi (_ BitVec 32))
(declare-const lo (_ BitVec 32))
(assert (not
    (=> (= x (concat hi lo))
        (= (bv2nat x)
           (+ (* 4294967296 (bv2nat hi))
              (bv2nat lo))))))
(check-sat)
```

```
(declare-const x (_ BitVec 64))
(declare-const hi (_ BitVec 32))
(declare-const lo (_ BitVec 32))
(assert (not
    (=> (= hi ((_ extract 63 32) x))
    (=> (= lo ((_ extract 31 0) x))
        (= (bv2nat x)
           (+ (* 4294967296 (bv2nat hi))
              (bv2nat lo)))))))
(check-sat)
```

Figure 3: Handcrafted example: concat and extract

```
(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(assert (not
        (=> (<= (+ (bv2nat x) (bv2nat y)) (bv2nat #xFFFFFFFF))
            (= (bv2nat (bvadd x y)) (+ (bv2nat x) (bv2nat y))))))
(check-sat)
```

Figure 4: Handcrafted example: add overflow

Table 3
Results of the manually crafted benchmarks (✓ indicates a successful proof, ✗ indicates a timeout)

Benchmark	Alt-Ergo (BV)	Z3	CVC5
Concat	✓	✓	✗
Extract	✓	✗	✗
Add overflow	✓	✗	✗

The last example, shown in Figure 4, is another simple property regarding the absence of overflow of integers when computing a bit-vector addition. As the `bvuaddo` operator from SMT-LIB 2.7 (used to detect when an addition overflows) is not supported by all solvers, we use a straightforward encoding using `bv2nat` instead using the standard's definition of the operator. This simple example is only proved by Alt-Ergo thanks to the communication between the integer and bit-vector theories.

The results on the handcrafted examples are summarised in Table 3.

5. Conclusion and Future Work

We have presented in this paper recent developments in the Alt-Ergo SMT solver. The main improvements are support for arithmetic and logical bit-vector operators from the SMT-LIB theory "BV", and the support for the SMT-LIB standard version 2.7 through Dolmen.

The implementation of arithmetic and logical bit-vector operators follows closely Chihani et al. [9], adapted to the context of a Shostak-style combination procedure. This was an opportunity to integrate a constraint propagation mechanism into Alt-Ergo's core solver, and a full rewrite of the intervals module for better modularity and readability.

In the near future, we plan to improve direct cross-theory propagations (notably between bit-vectors and arithmetic), and integrate the difference logic solver (called the global Delta domain in Chihani et al. [9]) in Alt-Ergo's main development branch. We also plan on reworking the case split mechanism in Alt-Ergo to better integrate with the CDCL solver, and on integrating Alt-Ergo's floating-point and bit-vector theories to provide a full implementation of the SMT-LIB floating-point theory.

Declaration on Generative AI

During the preparation of this work, the authors used Gemini 2.5 in order to: Peer review simulation. Further, the authors used ChatGPT for the abstract in order to: Improve writing style. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Preiner, H.-J. Schurr, C. Barrett, P. Fontaine, A. Niemetz, C. Tinelli, SMT-LIB release 2023 (non-incremental benchmarks), 2024. URL: https://doi.org/10.5281/zenodo.10607722. doi:10.5281/zenodo.10607722.

[2] S. Conchon, E. Contejean, J. Kanig, S. Lescuyer, CC(X): Semantical combination of congruence closure with solvable theories, in: Post-proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007), volume 198(2) of *Electronic Notes in Computer Science*, Elsevier Science Publishers, 2008, pp. 51–69. doi:10.1016/j.entcs.2008.04.080.

[3] S. Conchon, E. Contejean, M. Iguernelala, Canonized rewriting and ground AC completion modulo Shostak theories : Design and implementation, Logical Methods in Computer Science 8 (2012) 1–29. URL: http://www.lmcs-online.org/ojs/viewarticle.php?id=1037&layout=abstract&iid=40. doi:10.2168/LMCS-8(3:16)2012, selected Papers of the Conference *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 2011), Saarbrücken, Germany, 2011.

[4] S. Conchon, M. Iguernelala, K. Ji, G. Melquiond, C. Fumex, A Three-tier Strategy for Reasoning about Floating-Point Numbers in SMT, in: V. Kuncak, R. Majumdar (Eds.), Lecture Notes in Computer Science, volume 10427 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, 2017, pp. 419–435. URL: https://inria.hal.science/hal-01522770. doi:10.1007/978-3-319-63390-9_22.

[5] S. Conchon, A. Coquereau, M. Iguernlala, A. Mebsout, Alt-ergo 2.2, in: SMT Workshop: International Workshop on Satisfiability Modulo Theories, 2018.

[6] G. Bury, Dolmen: A validator for SMT-LIB and much more., in: SMT, 2021, pp. 32–39.

[7] G. Bury, F. Bobot, Verifying models with dolmen., in: SMT, 2023, pp. 62–70.

[8] G. Bury, Minimal logic detection and exporting SMT-LIB problems with dolmen (2024).

[9] Z. Chihani, B. Marre, F. Bobot, S. Bardin, Sharpening Constraint Programming approaches for Bit-Vector Theory, in: Integration of AI and OR Techniques in Constraint Programming, Integration of AI and OR Techniques in Constraint Programming, Padova, Italy, 2017. URL: https://cea.hal.science/cea-01795779.

[10] A. Zeljić, C. M. Wintersteiger, P. Rümmer, Deciding bit-vector formulas with mcsat, in: N. Creignou, D. Le Berre (Eds.), Theory and Applications of Satisfiability Testing – SAT 2016, Springer International Publishing, Cham, 2016, pp. 249–266. URL: http://www.philipp.ruemmer.org/publications/mcbv2016.pdf.

[11] D. Cyrluk, M. O. Möller, H. Rueß, An efficient decision procedure for the theory of fixed-sized bit-vectors, in: Proceedings of the 9th International Conference on Computer Aided Verification, CAV '97, Springer-Verlag, Berlin, Heidelberg, 1997, p. 60–71.

[12] L. D. Michel, P. Van Hentenryck, Constraint satisfaction over bit-vectors, in: M. Milano (Ed.), Principles and Practice of Constraint Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 527–543.

[13] J. Dougall, Bit-twiddling: Addition with unknown bits, 2020. URL: https://dougallj.wordpress.com/2020/01/13/bit-twiddling-addition-with-unknown-bits/.

[14] T. Feydy, A. Schutt, P. J. Stuckey, Global difference constraint propagation for finite domain solvers, in: Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP '08, Association for Computing Machinery, New York,

NY, USA, 2008, p. 226–235. URL: https://doi.org/10.1145/1389449.1389478. doi:10.1145/1389449.1389478.

[15] Y. Zohar, A. Irfan, M. Mann, A. Niemetz, A. Nötzli, M. Preiner, A. Reynolds, C. Barrett, C. Tinelli, Bit-precise reasoning via int-blasting, in: B. Finkbeiner, T. Wies (Eds.), Proceedings of the 23rd International Conference on Verification, Model Checking, and Abstract Interpretion (VMCAI'22), volume 13182 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 496–518. URL: http://theory.stanford.edu/~barrett/pubs/ZIM+22.pdf. doi:10.1007/978-3-030-94583-1_24.

[16] L. de Moura, N. Bjørner, Z3: an efficient smt solver, in: 2008 Tools and Algorithms for Construction and Analysis of Systems, Springer, Berlin, Heidelberg, 2008, pp. 337–340. URL: https://www.microsoft.com/en-us/research/publication/z3-an-efficient-smt-solver/.

[17] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, Y. Zohar, cvc5: A versatile and industrial-strength SMT solver, in: D. Fisman, G. Rosu (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, volume 13243 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 415–442. URL: https://doi.org/10.1007/978-3-030-99524-9_24. doi:10.1007/978-3-030-99524-9_24.

[18] A. Niemetz, M. Preiner, Bitwuzla, in: C. Enea, A. Lal (Eds.), Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II, volume 13965 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 3–17. URL: https://doi.org/10.1007/978-3-031-37703-7_1. doi:10.1007/978-3-031-37703-7_1.

[19] B. Dutertre, Yices 2.2, in: A. Biere, R. Bloem (Eds.), Computer-Aided Verification (CAV'2014), volume 8559 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 737–744.

[20] B. Marre, F. Bobot, Z. Chihani, Real Behavior of Floating Point Numbers, in: The SMT Workshop, SMT 2017, 15th International Workshop on Satisfiability Modulo Theories, Heidelberg, Germany, 2017. URL: https://cea.hal.science/cea-01795760.

[21] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, Why3: Shepherd your herd of provers, in: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland, 2011, pp. 53–64. https://hal.inria.fr/hal-00790310.