

# Alt-Ergo 2.2

Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, Alain Mebsout

▶ **To cite this version:**

Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, Alain Mebsout. Alt-Ergo 2.2. SMT Workshop: International Workshop on Satisfiability Modulo Theories, Jul 2018, Oxford, United Kingdom. hal-01960203

**HAL Id: hal-01960203**
**https://inria.hal.science/hal-01960203**

Submitted on 9 Jan 2019

**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.

# Alt-Ergo 2.2*

Sylvain Conchon[3,4], Albin Coquereau[2,3],
Mohamed Iguernlala[1,3], and Alain Mebsout[1]

[1] OCamlPro SAS, Gif-sur-Yvette, F-91190
[2] ENSTA ParisTech, Palaiseau, F-91120
[3] LRI, Université Paris Sud, CNRS, Orsay F-91405
[4] INRIA Saclay–Île-de-France, Orsay, F-91893

**Abstract**

Alt-Ergo is an SMT solver jointly developed by Université Paris-Sud and the OCamlPro company. The first version was released in 2006. Since then, its architecture has been continuously adapted for proving formulas generated by software development frameworks. As type systems with polymorphism arise naturally is such platforms, the design of Alt-Ergo has been guided (and constrained) by a native – and non SMT-LIB compliant – input language for a polymorphic first-order logic.

In this paper, we present the last version of Alt-Ergo, its architecture and main features. The main recent work is a support for a *conservative polymorphic* extension of the SMT-LIB 2 standard. We measure Alt-Ergo's performances with this new frontend on a set of benchmarks coming from the deductive program verification systems Frama-C, SPARK 2014, Why3 and Atelier-B, as well as from the SMT-LIB benchmarks library.

## 1 Introduction

Alt-Ergo 2.2 is the last version of the SMT solver developed by Université Paris-Sud and the OCamlPro company. In this paper, we present its architecture and main features. We also focus on a recent work for supporting a conservative *polymorphic* extension of SMT-LIB 2.

Since its first release in 2006, Alt-Ergo has been mainly designed for discharging proof obligations generated by software development frameworks. In particular, some of its features directly come from the Why/Why3 platforms for deductive program verification (also developed at Université Paris-Sud) which provide a rich specification language based on a *polymorphic* type system *à la* ML [11].

In order to directly handle proof tasks from this system, Alt-Ergo has a native input language for a polymorphic first-order logic and built-in capabilities to reason about parametric user-defined data-structures. The solver also supports quantifiers reasoning (based on E-matching), the free theory of equality, the theory of (integer and rational) arithmetic, enumerations, record data types and the theory of arrays. Recently, a procedure for the theory of floating-point arithmetic has been integrated. Last but not least, Alt-Ergo integrates a powerful mechanism for reasoning about associative-commutative function symbols.

The native input language of Alt-Ergo is not compliant with the SMT-LIB 2 standard and translating formulas from Alt-Ergo to SMT-LIB 2 (or vice-versa) is not immediate. Besides its extension with polymorphism, this native language diverges from SMT-LIB's by distinguishing terms of type `bool` from formulas (of type `prop`). This distinction makes it hard for instance to translate efficiently `let-in` and `if-then-else` constructs of the SMT-LIB.

*This work is partially supported by VOCaL (ANR-15-CE25-0008) and SOPRANO (ANR-14-CE28-0020) ANR projects.

In order to work closely with the SMT community, we have implemented in Alt-Ergo 2.2 a new frontend for a polymorphic conservative extension of the SMT-LIB 2 standard, similar to the one proposed by Bonichon *et al.* [4]. In this paper, we report on this last release and our first experiments with this new language.

The paper is organized as follows. In Section 2, we give an overview of the general architecture of Alt-Ergo 2.2 and present some of its key features:

- A purely applicative (or functional) implementation in OCaml
- A Shostak framework extended with AC completion for equational convex theories
- A conservative extension of SMT2 (PSMT2) to support prenex polymorphism à la ML
- A CDCL SAT solver tuned for simulating Tableau-like resolution method
- A collaborative framework for non-linear arithmetic based on interval calculus
- An interactive graphical user interface `AltGr-Ergo`

In Section 3, we give syntactic and semantic (typing) details about PSMT2. We also report on some issues to plug this new frontend in Alt-Ergo. Section 4 is dedicated to experimental evaluation. We consider benchmarks coming from deductive programs verification platforms and from the SMT-LIB library. We study Alt-Ergo's performances regarding several parameters like the benefit of polymorphism, the consequence of switching from its historical input language to a polymorphic SMT2 syntax, and the impact of using a CDCL solver instead of the Tableaux-like SAT engine. Finally, we discuss some future works in Section 5 and conclude.

## 2 General Overview and Key Features of Alt-Ergo 2.2

The general architecture of Alt-Ergo is depicted in Figure 1. Alt-Ergo is written in OCaml. Each module (rounded box in the picture) is implemented in a modular way as a set of (parameterized) modules. Most of the code (except very few parts like the CDCL algorithm and hashconsing used for maximal sharing) is written in a purely applicative programming style. According to our experimental evaluation (see Section 4), the performance loss inherent to functional programming is offset by the simplicity of use (in particular when backtracking) and the robustness offered by persistent data structures.

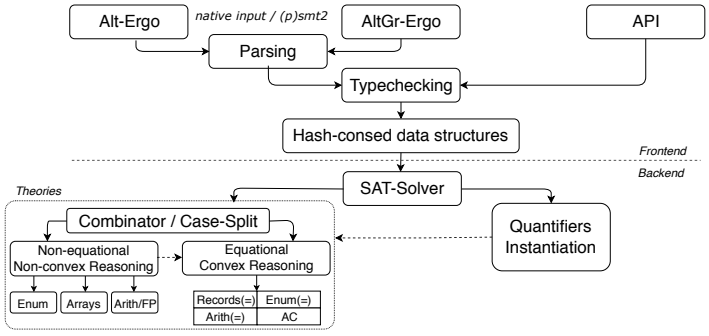



Figure 1: Overview of Alt-Ergo's architecture

Alt-Ergo provides decision procedures for reasoning in the combination of the following built-in theories: the free theory of equality with uninterpreted symbols, linear arithmetic over integers and rationals, fragments of non-linear arithmetic, polymorphic functional arrays with

extensionality, enumerated datatypes, record datatypes, associative and commutative (AC) symbols, floating-point arithmetic [9], and fixed-size bit-vectors with concatenation and extraction operators. Universal quantifiers are handled using the usual e-matching technique, extended to deal with type variables.

**Shostak combination.** One of the main modules of Alt-Ergo is the one which implements the *equational reasoning* for convex theories (equational convex parts of arithmetic, records, enumerations, ...). This algorithm, called CC(X), is reminiscent of Shostak combination: it maintains a union-find data-structure modulo a theory X, equipped with a *solver* and a *canonizer*, from which maximal information about implied equalities can be directly used for congruence closure [6]. This algorithm has been extended to handle associative and commutative user-defined symbols. The resulting combination method, called AC(X) [5], is obtained by augmenting in a modular way ground AC completion with the canonizer and solver present for the theory X. For instance, the first goal in Figure 2, which involves AC reasoning (symbol `u`), the free theory of equality (with the uninterpreted symbol `f`) and integer linear arithmetic, is automatically proved by Alt-Ergo.

**Non-linear arithmetic.** To reason about non-linear integer arithmetic, Alt-Ergo implements an algorithm which relies on the extension and collaboration of the AC(X) framework and interval calculus to handle NIA axioms in a built-in way [8]. An example requiring non-linear arithmetic reasoning is given in Figure 2. To prove such kind of goals, AC(X) is instantiated with linear integer arithmetic (LIA) to handle equalities of LIA and associativity and commutativity properties of non-linear multiplication. The interval calculus component of Alt-Ergo is used, in addition to standard linear operations over inequalities, to propagate bounds of non-linear terms and to suggest case-splits on finite domains.

```
(*+- example with an AC symbol -+-+-+-+-+-+-+-+*)
logic ac u : int, int -> int
logic f : int -> int
goal g :
   forall a,c1,c2,e1,e2,d,b : int.
   (u(a,c2 - c1) = a and b = e2 and
   u(e1,e2) - f(b) = u(d,d) and d = c1 + 1 and
   u(b,e1)=f(e2) and c2 = 2 * c1 + 1) ->
   a = u(a,0)

(*+- example with non-linear arithmetic +-+-+-+*)
logic v,t : int
goal g :
 forall w, x,y,z: int.
   v * t = 3 -> v * w = 5 ->
   -(y*y*y) + 3 * w - 5 * t  <= -10 ->
   0 <= x -> x <= 5 ->
   2 * z * ( x / y) + 3 * x = 4 ->
   3 * (x / y) * x <= 0 ->
   false
```

```
(*+- example with polymorphism +-+-+-+-+-+-+-+-*)
type 'a list

logic nil : 'a list
logic cons : 'a, 'a list -> 'a list
logic hd: 'a list -> 'a
logic tl: 'a list -> 'a list

axiom construction :
  forall l:'a list. l<> nil -> cons(hd(l),tl(l)) = l

axiom hd_cons :
  forall x:'a. forall l:'a list. hd(cons(x,l)) = x

axiom tl_cons :
  forall x:'a. forall l:'a list. tl(cons(x,l)) = l

goal g :
  forall l:'a list. forall r:('a list) list.
    r <> nil -> l = hd(r) -> l <> nil ->
    cons(cons(hd(l),tl(l)), tl(r)) = r
```

Figure 2: Three examples written in Alt-Ergo's native input language

**Polymorphism.** The historical input language of Alt-Ergo is a first-order logic with some built-in theories and polymorphic data types. Polymorphic types can appear in built-in theories or in user-defined data types. Alt-Ergo has a type discipline à la ML, which means that type variables are implicitly quantified by prenex universal quantifiers. For instance, the third

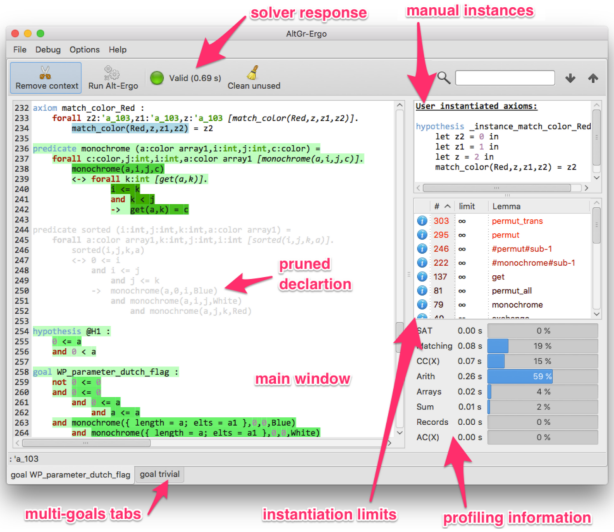
example in Figure 2 defines a polymorphic axiomatization for lists, where the goal formula combines two kinds of polymorphic lists. As explained in Section 3, we recently added a partial support for the SMT-LIB 2 standard, extended with ML-style prenex polymorphism.

**Tableaux-like and CDCL procedures.** Since its first versions, Alt-Ergo integrates a Tableaux like SAT solver modulo theories implemented in a purely functional programming style. This solver has many advantages in the context of deductive software verification: first, its search is guided by the shape of formulas that are not required to be in CNF form. In addition, it constructs "small" models compared to CDCL approach. For instance, given a formula $\varphi = (a \vee (b \vee \varphi_1)) \wedge (\neg a \vee (\neg b \vee \neg\varphi_1))$ where $\varphi_1$ is a non atomic sub-formula, the CDCL solver will not conclude that the assignment $\{a \to true, b \to false\}$ is a model for $\varphi$, contrary to the Tableaux-like approach.

Smaller models have two impacts: first, they allow to limit the number of calls to theories solvers (only literals that are true in the model are sent to theories solvers), and second, they help to manage efficiently the set of ground terms used by the E-matching engine to produce new instances from lemmas. Last but not least, the functional coding style of the Tableaux-like solver simplifies memory management of these instances after non chronological backtracking.

Despite its qualities, the Tableaux-like solver suffers from the lack of an efficient Boolean constraints propagation mechanism and a powerful clauses learning technique. As a result, it may have poor performances on SMT problems with complex Boolean structures. To overcome this issue, we recently worked on a new SAT solver that combines the efficiency of a CDCL engine with the nice properties of the Tableaux-like solver (construction of a small Boolean model, interaction with theories and instantiation engine using a small set of literals and terms). Section 4 gives experimental results for a comparison of these two approaches.

**Graphical User Interface.** To the best of our knowledge, Alt-Ergo is the only SMT solver equipped with a graphical user interface. The GUI allows to observe the triggers computed for each axiom, the number of instances produced per axiom, the time spent in reasoning engines, and the context that have been used to make a proof (if unsat-cores computation is enabled). One can also modify triggers or add instances manually, prune the context once a proof is done, or limit/disable some axioms. More details about AltGr-Ergo can be found in this paper [10].



# 3 Towards a Polymorphic SMT2 Extension in Alt-Ergo

In this Section, we discuss some aspects of our conservative extension of SMT2 to support prenex polymorphism à la ML. We show via small examples the modifications made in SMT2 to add parametric polymorphism, we present some typing rules of the extended language and some implementation details, and we discuss the main difficulties we faced when integrating the new frontend in Alt-Ergo.

## 3.1 Conservative extension of SMT2 language with polymorphism

**Syntax.** The syntax of the extension is inspired by the "*parameters notation*" introduced in the SMT-LIB language 2.6 for algebraic datatypes declaration, and is somehow similar to Bonichon *et al.* [4]. It consists in binding type variables with the "`par`" construct in some relevant SMT2 commands, and in using the type variables like any other type in the scope of "`par`". More precisely, we extended six SMT2 commands to obtain a polymorphic extension as shown in the examples of Figure 3. Original SMT examples are given on the left, and similar examples with polymorphism on the right.

```
(declare-datatype ip ((iP (il Int) (ir Int))))
(declare-const x Int)
(declare-fun f (Int Int) Int)
(assert (forall ((x Int)) (> x 1)))
(define-fun foo_f ((x Int)) Real (+ x x))
(define-fun-rec foo_h ((b Int)) Int) (foo_h b))

(define-funs-rec
    ((foo_u ((a Real) (b Int)) Int)
     (foo_v ((c Int) (d Real)) Int))
       ((foo_v b a) (foo_u d c)))
```

```
(declare-datatype p (par (A B) ((P (l A) (r B)))))
(declare-const y (par (A) A))
(declare-fun g (par (B C) (B C Int) B))
(assert (par (A) (forall ((x A) (y A)) (= x y))))
(define-fun foo_g (par (A B) ((x A)(y B)) B) (g y x 3))
(define-fun-rec foo_i (par (A) ((b A)) A) (foo_i b))

(define-funs-rec
    ((foo_w (par (X) ((a X) (b Int)) Int))
     (foo_x (par (Y) ((c Int) (d Y)) Int)))
       ((foo_x b a) (foo_w d c)))
```

Figure 3: Modified commands to support polymorphism in SMT2.

**Type-checking.** Type-checking PSMT2 files is done using destructive unification. The type system is similar to Why3's [11]. Some of the main rules are given in Figure 4. In these rules, a typing environment is a tuple $\langle\, \Gamma \,|\, \Delta \,\rangle$ in which $\Gamma$ a map from type symbols (including parameters) to arities, and $\Delta$ is a map from function symbols (and variables) to tuples $(par*, type*, type)$, where $par*$ denotes the (possibly empty) list of type parameters, and $type*$ denotes the (possibly empty) list of arguments types. We will write $\Gamma \vdash \tau$ to say that the type $\tau$ is well-formed, and write $\langle\, \Gamma \,|\, \Delta \,\rangle \vdash e : \tau$ to say that $e$ has type $\tau$ in the environment $\langle\, \Gamma \,|\, \Delta \,\rangle$. We use the notation $\langle\, \Gamma \,|\, \Delta \,\rangle \xrightarrow[\texttt{cmd}]{} \langle\, \Gamma' \,|\, \Delta' \,\rangle$ for the effect that a command `cmd` has on a configuration $\langle\, \Gamma \,|\, \Delta \,\rangle$. We assume that $\Gamma$ and $\Delta$ are initialized with some builtin type and function symbols, respectively[1]. We will distinguish between two kinds of type variables: parameters introduced with the "`par`" construct (denoted $A_1, \ldots, A_n$), and unification variables (denoted $\alpha_1, \ldots \alpha_n$) introduced during type-checking. We will sometimes use the vector notation for a list of elements. Note that, since we use destructive unification, the rules may have side-effects and the order of the premises is sometimes relevant.

The first rule updates $\Gamma$ after a sort declaration. The second (resp. third) one checks if a type or a parameter (resp. a unification variable) is well-formed. The fourth rule updates $\Delta$ with the signature of a function $f$ declared using `declare-fun`, possibly with type parameters. For an `assert` command (rule 5), we first add its type parameters, if any, before type-checking

[1] Depending on the logic that has been set with `set-logic` command.

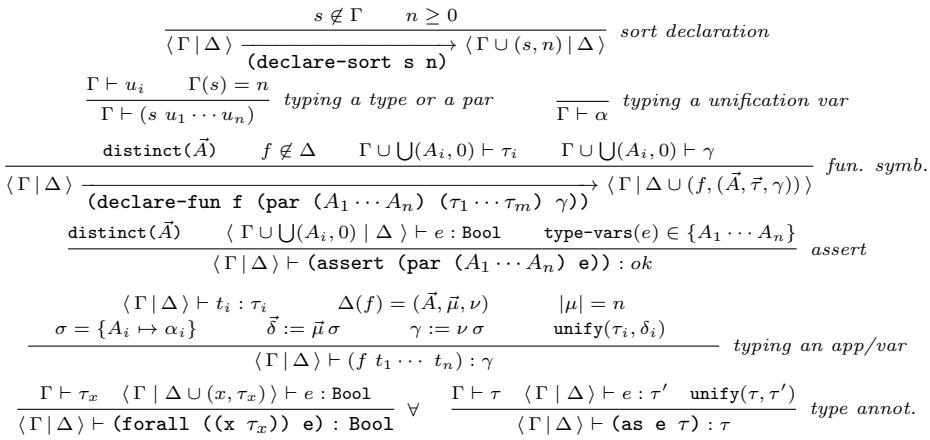



Figure 4: Some typing rules of the PSMT2 extension

the expression inside the assertion. Then, if $e$ is well typed, we check that the only type variables occurring in $e$ are those bound by `par` (i.e. all eventual unification variables are substituted).

Typing an application $(f\ t_1 \cdots\ t_n)$ is more subtle. Once its arguments are type-checked, we retrieve the declaration of $f$ from $\Delta$, and create a fresh signature $f : \vec{\delta} \to \gamma$. This is done by replacing every type parameter $A_i$ with a fresh unification variable $\alpha_i$. Once this is done, we unify every argument type $\tau_i$ with its corresponding type $\delta_i$. If all unifications succeed, we associate the type $\gamma$ to the application. It is important to note that the $\alpha_i$ are physically shared among $\delta_1, \cdots, \delta_n, \gamma$ to enable destructive unification, and hence the modification of $\gamma$ while unifying the $\tau_i$ against the $\delta_i$. Type-checking a variable is a particular case of the "application" case. The two latest rules are obvious.

Note that during the unification process, only "unification type variables" are eventually substituted. Type parameters behave likes "monomorphic" type constants and are never altered.

Similarly, the semantics of our PSMT2 language is similar to Why3's (also described in this paper [11]).

## 3.2 Implementation and issues

We implemented the PSMT2 extension as a standalone OCaml library, called `psmt2-frontend`[2]. Our work mainly consists in modifying an auto-generated parser[3] to support our syntax extension, and in implementing a type-checker that follows the lines of the typing rules given above. The library produces abstract syntax trees (ASTs) after parsing, and provides a type-checker that decorates different (sub-)expressions of a parsed AST with their respective types. It is currently at an early development stage and needs some improvements (support of Bit-vectors, Floating points, incremental mode, ...).

In a second step, we plugged the library in Alt-Ergo in order to support the PSMT2 extension. Currently, the integration consists in translating the type-checked AST produced by `psmt2-frontend` to Alt-Ergo's (parsed) abstract syntax tree. A better solution would be to directly generate a typed AST or advanced hash-consed data-structures. Unfortunately, since

[2] https://github.com/Coquera/psmt2-frontend

[3] http://homepage.divms.uiowa.edu/~astump/software/ocaml-smt2.zip

triggers are inferred during type-checking, we are obliged to type-check the AST again in Alt-Ergo.

The issue above is actually not the main difficulty we faced when translating `psmt2-frontend`'s AST to our solver. Indeed, until recently, Alt-Ergo had a very limited support for `let-in` and `if-then-else` (ITE) constructs. Since the solver makes a clear distinction between propositions and Booleans as outlined in Section 2, supporting all forms of ITE expressions[4] required non-trivial modifications in Alt-Ergo. More generally, the distinction between terms and formulas, prevents us from using propositions in user-defined function symbols, as shown in the example `(<= (f (or A B)) c)`. We partially addressed this issue by abstracting formulas that occur inside terms with fresh Boolean variables and introducing the abstraction with a `let-in`. For instance, the example above is lifted to `(let ((fresh_b (or A B))) (<= (f fresh_b) c))`. This transformation can be done during type-checking, which allows us to not modify existing hash-consed data-structures on terms, literals and formulas.

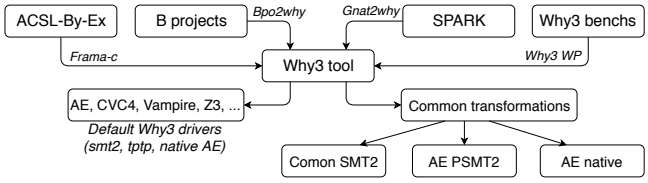
While our solution only slightly differs from Bonichon *et al.* [4] regarding syntax and type-checking, the two solutions completely diverge when it comes to effectively reason about polymorphism. In fact, they proposed a monomorphization approach while polymorphism is built-in in our instantiation engine. We refer the reader to this paper [2] for more details about some subtleties of polymorphism, and how it can be handled in an SMT-solver's back-end.

# 4 Experimental Evaluation

In this section, we evaluate Alt-Ergo on a set of benchmarks coming from programs verification. We will study Alt-Ergo's performances regarding several parameters like the benefit of polymorphism, the consequence of switching from its historical input language to a polymorphic SMT2 syntax, and the impact of using a CDCL solver instead of the Tableaux-like SAT engine. We also compare Alt-Ergo to some state-of-the-art (SMT) solvers on the benchmarks above, and on benchmarks taken from the SMT-LIB library. All benchmarks, StarExec jobs and detailed results are available here: https://gitlab.com/Coquera/SMT-Workshop--2018.

## 4.1 Benchmarks from program verification

We will consider four sources of benchmarks: SPARK programs from *AdaCore*'s github, C programs taken from *Fraunhofer Fokus*' ACSL by Example project, Why3's gallery of verified programs, and four industrial B projects that were previously discharged automatically or interactively with Atelier-B. Two of these projects (called DAB and RCS3) were provided by *Mitsubishi Electric R&D Centre Europe*. Two additional benchmarks (called p4 and p9) were provided by *ClearSy*.




All these benchmarks are translated to Why3. Formulas are then produced in different input formats using Why3's *drivers* technology [3], as shown in Figure above. First, we use the

[4] Like, for instance, `(<= x (ite (A or B) 1 (+ y 2)))`

*default drivers* shipped with Why3 to (optimally) translate the formulas for every considered solver. In a second step, we apply some transformations to the Why3 files (such as axiomatization of bit-vectors, and floating-point arithmetic, which are not supported by all considered SMT solvers). The goal of this step is to be as fair as possible when comparing results of different solvers. Once the transformations applied, we produce benchmarks using three different drivers: (1) a common SMT2 driver that is basically the CVC4's default driver with the previous additional transformations, (2) a modified version of driver 1 that disables0 polymorphism encoding, allowing us to obtain benchmarks in PSMT2 format, (3) a modified version of driver 2 that outputs formulas in Alt-Ergo's native syntax instead of PSMT2. At the end we obtain benchmarks with seven different encoding: default Alt-Ergo, default CVC4, default Z3, default vampire, common SMT2, common PSMT2, and common Alt-Ergo.

We ran Alt-Ergo 2.2.0, CVC4 1.5, Z3 4.6.0, and Vampire 4.2.2 on these benchmarks on the StarExec platform. Time limit was set to 60 seconds and memory limit to 10 GB. We report in the tables below the number of proved formulas (p), the cumulative time (t) taken to prove them, and the percentage (%). For the graphs, x-axes represent the number of unsat answers and y-axes cumulative time in seconds. Note that, we are mainly interested in unsatisfiability (or dually in validity) in the context of program verification. So, *"proved"* here means *"unsat"* (or *"Valid"*).

**Impact of polymorphism and of the SAT-solver engine.** The table and the graph in Figure 5 show the results obtained when running Alt-Ergo on different benchmarks in the *common* category. We notice that: (1) in general, the CDCL solver competes equally (or is sometimes better) than the Tableaux-like engine regarding the number of proved formulas and resolution time, (2) resolution rate and resolution time are better when using PSMT2, compared to standard SMT format, (3) for some benchmarks (eg. Why3), Alt-Ergo is better on our native polymorphic language compared to PSMT2, (4) for the SPARK benchmark, we notice that there is not a real difference when comparing results on SMT2 and PSMT2 encoding. Indeed, there is not a lot of polymorphism in these benchmarks.

To sum up, Alt-Ergo is better on a polymorphic input syntax, in particular regarding resolution time. In addition, even if the PSMT2 frontend is new and needs some additional improvements, the results we obtain with it are quite close to those we get with the historical native input language. Moreover, the newly added CDCL solver is better than the old Tableaux-like engine.

| common | | smt2 | smt2 | psmt2 | psmt2 | native | native |
|---|---|---|---|---|---|---|---|
| | | tab | cdcl | tab | cdcl | tab | cdcl |
| why3 | p | 1243 | 1221 | 1345 | 1326 | 1356 | 1343 |
| (2003) | t | 1454 | 1065 | 1481 | 694 | 1156 | 1211 |
| spark | p | 14027 | 14033 | 14025 | 14032 | 13978 | 14028 |
| (16773) | t | 1650 | 1275 | 1671 | 1301 | 2581 | 2163 |
| acsl | p | 852 | 842 | 897 | 891 | 897 | 888 |
| (959) | t | 321 | 173 | 198 | 303 | 312 | 213 |
| rcs3 | p | 2224 | 2225 | 2227 | 2227 | 2229 | 2227 |
| (2256) | t | 1780 | 1575 | 939 | 919 | 609 | 575 |
| dab | p | 785 | 816 | 840 | 840 | 838 | 840 |
| (860) | t | 768 | 485 | 872 | 107 | 722 | 33 |
| p4 | p | 8876 | 9097 | 9282 | 9303 | 9288 | 9303 |
| (9341) | t | 8876 | 6668 | 3113 | 1937 | 2704 | 1039 |
| p9 | p | 232 | 219 | 247 | 282 | 278 | 282 |
| (371) | t | 1393 | 1631 | 353 | 1055 | 410 | 985 |
| | p | 28239 | 28453 | 28863 | 28901 | 28864 | 28911 |
| total | % | 87.1 | 87.8 | 89.0 | 89.2 | 89.0 | 89.2 |
| (32563) | t | 16262 | 12876 | 8631 | 6319 | 8496 | 6222 |

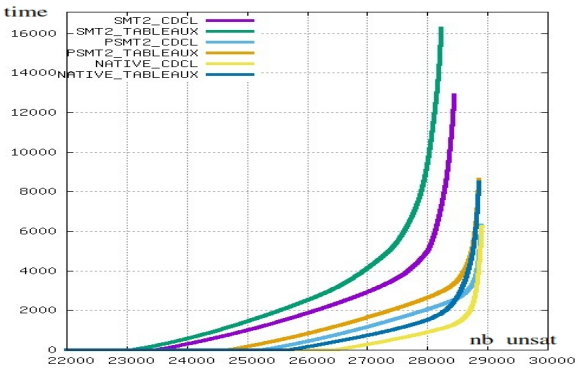



Figure 5: Comparing Alt-Ergo with different SAT-solvers and on different input languages

**Fair comparison on SMT2 benchmarks.** The table and the graph in Figure 6 show the results obtained when running different solvers on the same SMT2 input files. For readability, y-axis is cropped at 25,000 seconds. We remark that Alt-Ergo performs well in general. This was quite surprising for us, knowing that we mainly rely on functional OCaml data-structures and we didn't extensively investigated their improvement. When looking to the table in more details, we notice that CVC4 and Z3 are really fast on some benchmarks. For instance, this is the case on Why3 and SPARK for CVC4, and on SPARK, RCS3 and p9 for Z3.

| common | | smt2 tab | smt2 cdcl | smt2 cvc4 | smt2 z3 | smt2 vam |
|---|---|---|---|---|---|---|
| why3 (2003) | p | 1243 | 1221 | 1196 | 1193 | 899 |
| | t | 1454 | 1065 | 527 | 2161 | 2518 |
| spark (16773) | p | 14027 | 14033 | 14010 | 13821 | 12690 |
| | t | 1650 | 1275 | 703 | 352 | 5459 |
| acsl (959) | p | 852 | 842 | 686 | 709 | 506 |
| | t | 321 | 173 | 106 | 400 | 2122 |
| rcs3 (2256) | p | 2224 | 2225 | 2039 | 2175 | 1166 |
| | t | 1780 | 1575 | 2012 | 270 | 27488 |
| dab (860) | p | 785 | 816 | 503 | 722 | 459 |
| | t | 768 | 485 | 352 | 110 | 4088 |
| p4 (9341) | p | 8876 | 9097 | 8994 | 7861 | 2507 |
| | t | 8893 | 6668 | 8894 | 3879 | 46459 |
| p9 (371) | p | 232 | 219 | 243 | 238 | 126 |
| | t | 1393 | 1631 | 1026 | 490 | 2315 |
| total (32563) | p | 28239 | 28453 | 27671 | 26719 | 18353 |
| | % | 87.1 | 87.8 | 85.4 | 82.4 | 56.6 |
| | t | 16262 | 12876 | 13122 | 7666 | 90453 |

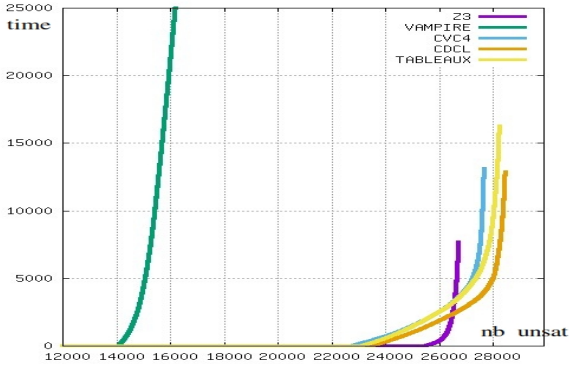



Figure 6: Comparing Alt-Ergo with other solvers on the same SMT2 input files

**Comparing solvers on files obtained with their default drivers.** The table and the graph in Figure 7 show the results obtained when running different solvers on files obtained with their respective drivers shipped with Why3. As expected, all solvers perform better on these benchmarks compared to table of Figure 6. Moreover, Alt-Ergo with CDCL additionally improves its resolution time significantly. Indeed, total resolution time with its default driver is smaller than those of both Figure 5 and Figure 6. We also remark that CVC4 and Z3 are really good on SPARK benchmarks. Actually, these benchmarks heavily involve Ada "Integer range types" that are encoded as bit-vectors for these solvers. Disabling this transformation in the "common SMT2" benchmark seems to have a negative impact on them.

| default | | native tab | native cdcl | smt2 cvc4 | smt2 z3 | tptp vam |
|---|---|---|---|---|---|---|
| why3 (2003) | p | 1363 | 1347 | 1320 | 1188 | 664 |
| | t | 1110 | 1320 | 810 | 2036 | 8768 |
| spark (16773) | p | 14012 | 14045 | 14198 | 14373 | 12604 |
| | t | 2500 | 1788 | 540 | 1203 | 39290 |
| acsl (959) | p | 901 | 892 | 895 | 740 | 438 |
| | t | 191 | 213 | 362 | 525 | 5171 |
| rcs3 (2256) | p | 2235 | 2233 | 2040 | 2182 | 1129 |
| | t | 614 | 575 | 2774 | 104 | 3675 |
| dab (860) | p | 859 | 859 | 812 | 696 | 492 |
| | t | 806 | 30 | 1158 | 141 | 8488 |
| p4 (9341) | p | 9272 | 9291 | 9105 | 7900 | 4047 |
| | t | 2288 | 1047 | 9882 | 2141 | 85125 |
| p9 (371) | p | 278 | 275 | 187 | 231 | 88 |
| | t | 456 | 643 | 934 | 432 | 3425 |
| total (32563) | p | 28920 | 28942 | 28557 | 27310 | 19462 |
| | % | 89.2 | 89.3 | 88.1 | 84.3 | 60.0 |
| | t | 7968 | 5619 | 16462 | 6585 | 153945 |

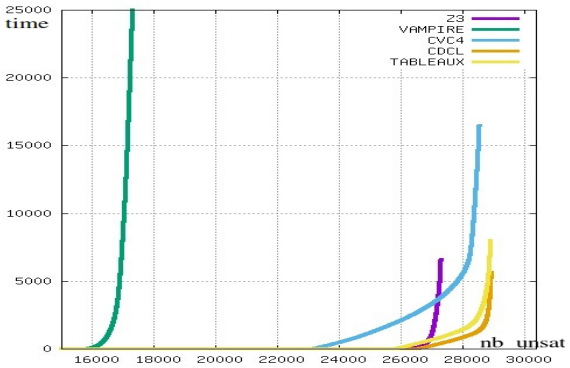



Figure 7: Comparing solvers on files obtained with their respective drivers

### 4.2 Benchmarks from SMT-LIB

For the evaluation of our new front-end, we selected the AUFLIRA and AUFNIRA categories from the SMT-LIB library[5]. Their formulas are, to the best of our knowledge, the closest to deductive programs verification. The result [6] are given in Figure 8. Overall, although Alt-Ergo (with both a CDCL-based and with a Tableaux-like SAT solver) is behind the other solvers, the gap is not that big. We were actually positively surprised, as we expected worse resolution rate before benchmarking. A more detailed look at AUFLIRA results shows that Alt-Ergo is performing poorly on *peter* and *why* sub-directories. A quick analysis revealed that this is mainly due to triggers inference/instantiation, in particular in presence of `let-in` constructs.

| AUFLIRA | | tab | cdcl | cvc4 | z3 | vam |
|---|---|---|---|---|---|---|
| why (1271) | p | 1202 | 1209 | 1233 | 1242 | 1243 |
| | t | 2446 | 778 | 243 | 1 | 3411 |
| nasa (18446) | p | 18286 | 18286 | 18390 | 18405 | 18182 |
| | t | 26 | 39 | 8 | 1 | 1167 |
| peter (198) | p | 1 | 1 | 77 | 100 | 7 |
| | t | ∼0 | ∼0 | 1368 | 33 | 13 |
| FFT (94) | p | 2 | 3 | 4 | 4 | 2 |
| | t | 3 | 7 | 46 | 5 | 11 |
| total (20011) | p | 19491 | 19499 | 19704 | 19751 | 19434 |
| | % | 97.4 | 97.4 | 98.5 | 98.7 | 97.1 |
| | t | 2476 | 825 | 1666 | 39 | 4603 |

| AUFNIRA | | tab | cdcl | cvc4 | z3 | vam |
|---|---|---|---|---|---|---|
| why (13) | p | 13 | 13 | 13 | 13 | 13 |
| | t | ∼0 | ∼0 | ∼0 | ∼0 | 19 |
| nasa (976) | p | 936 | 936 | 951 | 944 | 945 |
| | t | ∼0 | ∼0 | 107 | ∼0 | 94 |
| aviation (21) | p | 8 | 7 | 8 | 8 | 20 |
| | t | ∼0 | ∼0 | ∼0 | ∼0 | 217 |
| FFT (470) | p | 51 | 55 | 61 | 58 | 42 |
| | t | 4 | 89 | 78 | 33 | 211 |
| total (1480) | p | 1008 | 1011 | 1033 | 1023 | 1020 |
| | % | 68.1 | 68.3 | 69.8 | 69.1 | 68.9 |
| | t | 4 | 89 | 186 | 33 | 542 |

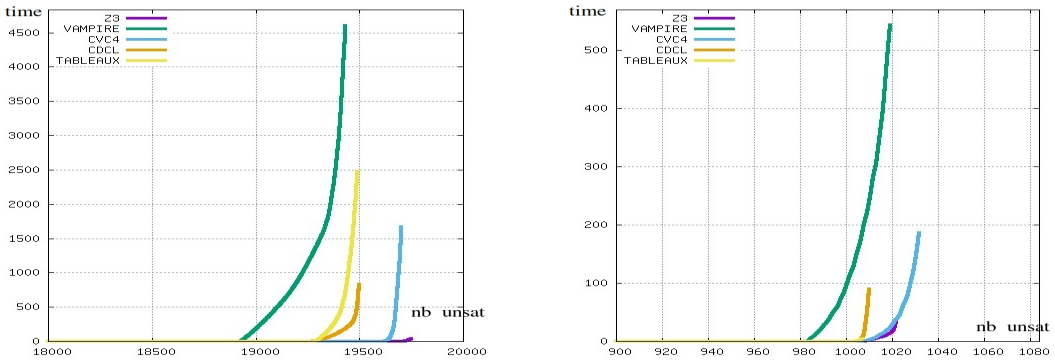



Figure 8: Results on AUFLIRA and AUFNIRA: only unsat answers are reported, the % is computed w.r.t. total number of formulas in each directory. X-axes are zoomed in the graphs.

## 5 Conclusion and Future Works

We have presented in this paper the last version of the Alt-Ergo SMT solver. One of the main recent work is a frontend for the SMT-LIB 2 format with a non intrusive extension to support prenex polymorphism.

The evaluation of Alt-Ergo on a test-suite coming from deductive programs verification showed that our solver is better on polymorphic input formulas, compared to the *monomorphized* versions of these inputs. Using these benchmarks, and others coming from the SMT-LIB library, we also noticed a gain of performance when switching from our old Tableaux-like SAT solver to a new CDCL-based engine.

[5] See here http://smtlib.cs.uiowa.edu/logics.shtml for the meaning of these categories.
[6] time limit = 60 seconds, memory limit = 10 GB, only 'unsat' answers are reported.

In addition to polymorphism, Alt-Ergo is the only SMT solver that is equipped with a graphical user interface. It is also one of the latest SMT solvers that is (still) using a Shostak-like procedure to combine theories. A modified version of it (called Alt-Ergo-Zero) is used inside the Cubicle model-checker [7], and the release 0.95.2 has been qualified by *Airbus Industrie* to be used as a backend for the Caveat platform [1].

In the near future, we plan to improve our psmt2-fronted library as well as its integration in Alt-Ergo. Our ultimate plan is to make PSMT2 the default input language of Alt-Ergo. But, before that, we should consider all the engineering issues discussed in Section 3. Moreover, since we now fully support `let-in` and `if-then-else` constructs, we should handle them efficiently, and eventually add some pre-processing in the solver. Finally, it would be interesting to see how to map SMT-LIB's floating-point arithmetic to Alt-Ergo's theory.

**Acknowledgment.** We thank the anonymous reviewers for their remarks which helped us improve this paper. We also thank AdaCore for providing SPARK benchmarks in Why3 format.

# References

[1] Patrick Baudin, Anne Pacalet, Jacques Raguideau, Dominique Schoen, and N. Williams. Caveat: a tool for software validation. In *Proceedings International Conference on Dependable Systems and Networks*, pages 537–, 2002.

[2] François Bobot, Sylvain Conchon, Evelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In *SMT '08: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories*, pages 1–5, 2008.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. 05 2012.

[4] Richard Bonichon, David Déharbe, and Cláudia Tavares. Extending SMT-LIB v2 with $\lambda$-Terms and Polymorphism. In Philipp Rümmer and Christoph M. Wintersteiger, editors, *Proceedings of the 12th International Workshop on Satisfiability Modulo Theories, SMT 2014*, volume 1163, 2014.

[5] Sylvain Conchon, Evelyne Contejean, and Mohamed Iguernelala. Canonized Rewriting and Ground AC Completion Modulo Shostak Theories : Design and Implementation. *Logical Methods in Computer Science*, 8(3), 2012.

[6] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X): Semantic Combination of Congruence Closure with Solvable Theories. *Electronic Notes in Theoretical Computer Science*, 198(2):51–69, May 2008.

[7] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 718–724, 2012.

[8] Sylvain Conchon, Mohamed Iguernelala, and Alain Mebsout. A collaborative framework for non-linear integer arithmetic reasoning in alt-ergo. In *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013*, pages 161–168, 2013.

[9] Sylvain Conchon, Mohamed Iguernlala, Kailiang Ji, Guillaume Melquiond, and Clément Fumex. A Three-Tier Strategy for Reasoning About Floating-Point Numbers in SMT. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 419–435, 2017.

[10] Sylvain Conchon, Mohamed Iguernlala, and Alain Mebsout. AltGr-Ergo, a Graphical User Interface for the SMT Solver Alt-Ergo. In *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016.*, pages 1–13, 2016.

[11] Jean-Christophe Filliâtre. One Logic To Use Them All. In Maria Paola Bonacina, editor, *CADE 24 - the 24th International Conference on Automated Deduction*, June 2013.