AltGr-Ergo, a Graphical User Interface for the SMT Solver Alt-Ergo

Sylvain Conchon
LRI, Université Paris-Sud
Orsay, France
conchon@lri.fr

Mohamed Iguernlala
OCamlPro SAS
Gif-sur-Yvette, France
mohamed.iguernlala@ocamlpro.com

Alain Mebsout
The University of Iowa
Iowa City, USA
alain-mebsout@uiowa.edu

Due to undecidability and complexity of first-order logic, SMT solvers may not terminate on some problems or require a very long time. When this happens, one would like to find the reasons why the solver fails. To this end, we have designed AltGr-Ergo, an interactive graphical interface for the SMT solver Alt-Ergo which allows users and tool developers to help the solver finish some proofs. AltGr-Ergo gives real time feedback in order to evaluate and quantify progress made by the solver, and also offers various syntactic manipulation options to allow a finer grained interaction with Alt-Ergo. This paper describes these features and their implementation, and gives usage scenarios for most of them.

1 Introduction

Alt-Ergo is an SMT solver designed for checking logical formulas generated by deductive program verification frameworks. For instance, Alt-Ergo is used as a back-end in the Why3 plateform [9]. It is also used to discharge formulas derived from C programs in Frama-C [11], from Ada programs in SPARK [2] or from B machines in Atelier-B [4, 7].

The Alt-Ergo input files produced by such tools share the same structure. They start with a prelude that contains a set of definitions (datatypes and logical symbols) and axioms for the encoding of theories specific to program verification (complex data structures, memory models, etc.). The rest of the file contains a proof obligation (PO) generated by a weakest precondition calculus.

Alt-Ergo checks such input files by the use of a combination of decision procedures (SAT, simplex, congruence closure, etc.) for reasoning about builtin theories (Booleans, arithmetics, equality, etc.) and a matching algorithm for instantiating quantified formulas.

Due to undecidability of first-order logic, Alt-Ergo may not terminate on some problems. When this happens, one would like to find the reasons why the solver fails. Most of the time, Alt-Ergo is either overwhelmed by a huge number of useless instances of axioms (causing a high activity in its decision procedures), or it fails to produce the good instances of lemmas that are mandatory to prove a goal.

A possibility for inspecting the internals of the solver is to output debugging information. This is however impractical because there is simply too many things to display and the output rapidly becomes unreadable. To help users (or developers) find out and understand what is going on, we have designed AltGr-Ergo, a graphical user interface for Alt-Ergo. As shown in Figure 1, our GUI displays at runtime crucial profiling information about the internal activities of the solver (time spent in decision procedures, number of instantiations, etc.). Some interaction features have also been added so that one can manually help the solver prove a goal (manual instances of lemmas, selection of hypotheses, etc.).

The main features of AltGr-Ergo are described in the next sections. The interface can display the following profiling information:

- unsat cores (Section 3.1)

- number of instances produced by axiom (Section 3.2)

- time spent in decision procedures (Section 3.3)

Interactive features include the following syntactic manipulations:

- pruning operations (Sections 4.1 and 4.2) for deactivating some part of the prelude;

- manual instances of lemmas (Section 4.3);

- selection and modification of triggers (Section 4.4) to change the heuristics used to guide the matching algorithm.

Last but not least, AltGr-Ergo provides a session mechanism (Section 4.5) which allows a user to save and replay all his modifications (selections, manual instances, etc.) on a given problem.
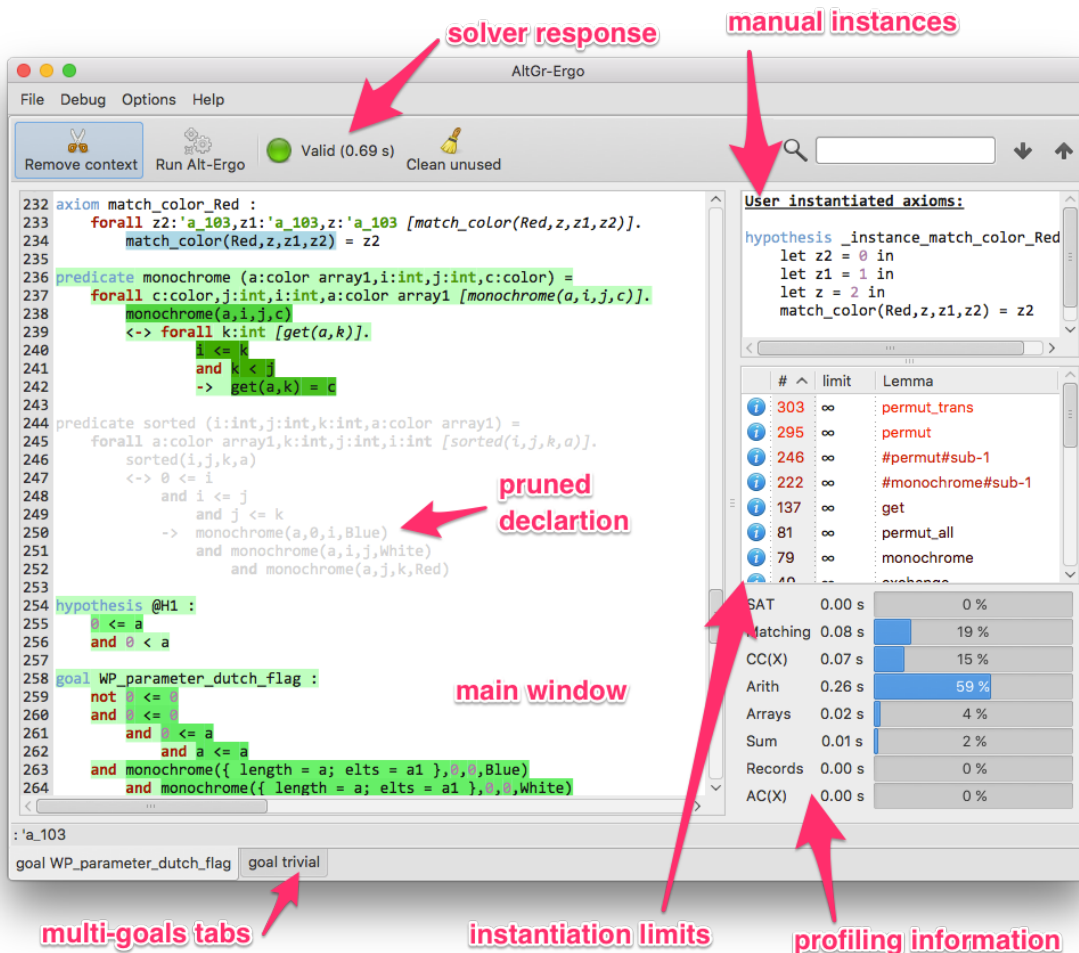

Figure 1: Overview of AltGr-Ergo's interface

2   A Short Introduction to Alt-Ergo

In order to understand some aspects of the graphical interface, we briefly present in this section Alt-Ergo's syntax and a high level overview of its main components.

The input language of Alt-Ergo is an extension of first-order logic with builtin theories and prenex polymorphism[1] *à la* ML [3]. Figure 2 shows a small proof obligation written in Alt-Ergo's syntax: first, a type symbol set parameterized by a type variable α is declared. Then, a polymorphic function (*resp.* predicate) symbol add (*resp.* mem) is introduced. After that, an axiom mem_add that gives the meaning of membership over the add symbol is stated. Finally, two integer symbols a and b, two sets of integers symbols s1 and s2, and a goal are given.

In addition to the Boolean connective "→", the "toy goal" mixes symbols from two theories: the free theory of equality (mem, add, a, b, ...), and linear arithmetic (+, -, 1). It is made of two parts: the hypotheses a = b + 1 and s2 = add(b,s1), and the conclusion mem(a - 1, add(b,si)) we would like to prove valid. Thanks to the second hypothesis and a *ground instance* of axiom mem_add (where x is replaced by a - 1, y by b, s by s1 and α by int), the conclusion is equivalent to (a-1 = b ∨ mem(a-1, s1)). Moreover, the latter formula always holds because a - 1 = b is equivalent to the first hypothesis modulo linear arithmetic. We thus conclude that the goal is valid.

```
type α set

logic add: α, α set → α set
logic mem: α, α set → prop

axiom mem_add:
  ∀ x, y: α. ∀ s: α set.
    mem(x, add(y, s)) ↔ (x = y ∨ mem(x, s))

logic a, b: int
logic s1, s2: int set

goal g:
  a = b + 1 →
  s2 = add(b,s1) →
  mem(a - 1, s2)
```

Figure 2: An example problem in Alt-Ergo's syntax

Alt-Ergo handles such proof obligations following the architecture given in Figure 3. The solver can be called either via its command-line "alt-ergo" or via its graphical user interface "altgr-ergo". The front end provides some basic operations such as parsing, type-checking, triggers inference[2] and translation of input formulas to richer data structures manipulated by back end modules.

The SAT solver plays a central role in Alt-Ergo. Given a formula, it tries to build a (partial) Boolean model for the ground part that is neither contradicted by the decision procedures, nor by the instances generated from (universally quantified) axioms. Its main operations are guessing truth values of (immediate) sub-formulas appearing in disjunctions (*decision*) and propagating unit facts that have been deduced (*bcp*).

---

[1]Type variables, if any, are prenex and implicitly universally quantified.
[2]this notion is crucial to control how axioms are instantiated, and is explained at the end of this section
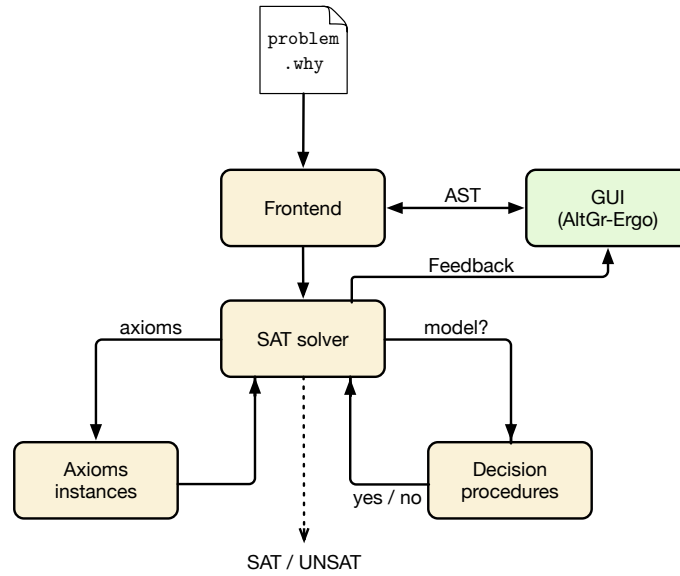
Figure 3: Alt-Ergo's simplified architecture

Atomic formulas (literals) are sent to "decision procedures" to check if they are consistent in the union of supported theories, and universally quantified formulas are sent to an "axioms instantiation" engine. If an inconsistency that does not involve any decision is detected, the given goal is valid[3]. Otherwise, when the SAT reaches a fix-point (*i.e.* succeeds in building a Boolean model), it asks the "axioms instantiation" part for some new ground instances. These instances are added to the SAT's context and reasoning continues.

Decision procedures component provides a combination of decision algorithms for a collection of built-in theories. Alt-Ergo supports some theories that are useful in the context of program verification, such as 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 and record datatypes, and associative and commutative (AC) symbols. More details of our combination techniques, which are not necessary to understand the rest of the paper, can be found here [6, 5, 10].

To reason about axioms, Alt-Ergo uses an instantiation mechanism based on E-matching [12] techniques. It generates ground consequences from assumed axioms based on some heuristics and information provided by the SAT solver and the decision procedures. The challenge is to heuristically produce useful instances that will allow to discard the current SAT's model, thus reducing the search space, and hopefully derive unsatisfiability (validity).

In addition to axioms, the instantiation engine requires a set of ground terms, and their partition into equivalence classes (computed by the decision procedures). In general, considered ground terms are those that appear in the decision procedures environment when instantiating. If this does not generate any instance, all the ground terms that appear in the current Boolean model are considered. Another key ingredient is the use of the notion of triggers (*a.k.a* patterns or filters) to guess which instances may be relevant depending on the SAT and decisions procedures' context.

A trigger for an axiom $\psi \equiv \forall \vec{x}.\phi(\vec{x})$ is a term (or a set of terms) that usually appears in $\phi(\vec{x})$ and which contains all the (term) variables $\vec{x}$ and all the type variables in $\phi(\vec{x})$. We use the notation

$$\forall \vec{x} \, [\, p \mid p_1, p_2]. \, \phi(\vec{x})$$

---

[3]To prove validity, Alt-Ergo internally assumes the negation of the conclusion and tries to deduce unsatisfiability.

to indicate that ψ is associated with one mono-trigger $\{p\}$, and one multi-trigger $\{p_1, p_2\}$. Triggers can either be provided by the user with the syntax above, or heuristically computed by Alt-Ergo. In the latter case, Alt-Ergo will choose at most two triggers per axiom by default. For instance, possible triggers for the axiom `mem_add` of Figure 2 are:

$$\{\texttt{mem(x, add(y, s)) }\} \qquad \{\texttt{add(y, s), add(x, s)}\} \qquad \cdots \qquad \{\texttt{x, y, s}\}$$

The latest multi-trigger is a very bad choice and is never selected by Alt-Ergo. In fact, it would generate an instance of `mem_add` for every (well-typed) combination of terms appearing in the decision procedures (*resp.* SAT's model). The two first triggers seem to be good choices. However, only the first one will permit us to prove the validity of the example in Figure 2. Indeed, the ground term `mem(a - 1, s2)` *matches* the trigger `mem(x, add(y, s))` modulo the equality `s2 = add (b, s1)`. The E-matching process produces the substitution $\{\texttt{x} \mapsto \texttt{a} - 1, \texttt{y} \mapsto \texttt{b}, \texttt{s} \mapsto \texttt{s1}, \alpha \mapsto \texttt{int}\}$, which allows us to generate the needed instance.

The rest of the paper describes the features (and their implementation) that AltGr-Ergo offers and shows how they can be useful both from an end-user perspective as well as from a developer's perspective.

3   Feedback

The first purpose of AltGr-Ergo is to provide *feedback* which can be useful at times to understand and evaluate what is happening inside the solver. Feedback is useful for users as a visual aid to make sense of the solver's progress, but it is also a precious tools for developers to profile and debug the solver.

3.1   Unsat Cores and Minimal Context Extraction

An *unsatisfiable core* in SMT, is a subset of the input formulas that make the problem unsatisfiable. Traditionally, SMT solvers will return sets where the elements are some of the input, top-level formulas, identified by a unique name in the source. Alt-Ergo goes a bit further and identifies sub-formulas that arise from the CNF (conjunctive normal form) conversion. This allows to identify more precisely which part of the formula is actually useful in proving the goal.

Unsat cores production is deactivated by default when running Alt-Ergo, but the interface offers a way to change solver options on the fly, even while the solver is running. In text, mode Alt-Ergo will spit unsat cores as pretty-printed formulas on its output. This can become large at times. The interface will display unsat cores in a more user-friendly way, visually identifying useful parts of the context, hypotheses, *etc.* by highlighting them in green (see Figure 1 for instance).

Different shades of green are used to highlight unsat cores in the buffer window. Top-most declarations and definitions which contain part of the unsat core will be highlighted in the lightest green and (sub-) formulas that appear more frequently in the unsat core will be highlighted with a darker shade. In particular, if an axiom is instantiated several times and the same part of the resulting instances is actually useful to prove the goal, then the user will be able to see this information visually. These features make it easy to rapidly identify which parts of the context are useful and how crucial they are to prove the goal.

Unsat cores also serve another purpose. By identifying which part of the context is effectively used to prove the goal by the solver, we can remove any other information contained in the problem while still having the guarantee that the goal will be provable by the solver[4]. AltGr-Ergo offers a button in the

---

[4]This guarantee might be lost in some particular cases in Alt-Ergo, namely when an instance of a useless quantified axiom is used as a *source of terms* to trigger the instantiation of another useful axiom.

toolbar to quickly remove every top-level declaration or definition that does not participate in the unsat core. Coupled with the mechanism of sessions (see Section 4.5), this allows to save and replay already proven goals much more rapidly.

3.2   Instantiation

As remarked in the introduction, quantified formulas are a source of incompleteness and inefficiencies in most SMT solvers. Providing a way to accurately and concisely expose information about instantiation is important for the user experience. AltGr-Ergo does so by displaying axioms instantiated, in real time, in a sub-window of the interface as shown in Figure 4.

Figure 4: Instances and manual limits

Here we report the number of instances produced by each axiom. They are listed in decreasing order of number of instances and their names[5] are colored in varying shades of red to denote frequency of instantiation. Axioms whose name is of a more saturated red denote the ones which produce more instances with respect to the total number of instances (regardless of its origin) generated at this point in time by the solver. This allows to quickly identify potentially problematic axioms which generate too many ground instances. This feedback gives indication regarding the likely cause of problems in the instantiation mechanism.

When this happens, we also offer the possibility to *limit* instantiation of particular quantified lemmas. For example, the left report of Figure 4 tells us that the lemma permut_exists was instantiated 1322 times, more than twice as much as any other lemma. This is thus a good candidate to limit instantiation. On the right screen capture of Figure 4, we limited the instances of this problematic lemma to 200 and an associated lemma (permut) to 300 (we performed this process iteratively, by first limiting instances of permut_exists and looking at what other lemmas were problematic). Lemmas for which instantiation has reached its given limit are shown in blue. We can notice that the runtime of the solver is reduced subsequently by a factor 3 for this particular example.

---

[5]For lemmas that are nested in larger formulas, we report the top-level name with some indication of their position. However, users can access their corresponding location in the source code by simply double clicking on the displayed name.

3.3 Profiling

Much like in the spirit of the previous section, the bottom-right most sub-window of the interface (see Figure 1) gives *real time* profiling information for the different modules and theories of Alt-Ergo. These include the time spent in the SAT solver, the matching procedure, the congruence closure algorithm (CC(X)), the builtin support for associative and commutative symbols (AC(X)) [5] and the theories of arithmetic, arrays, enumerated data-types (Sum) and records.

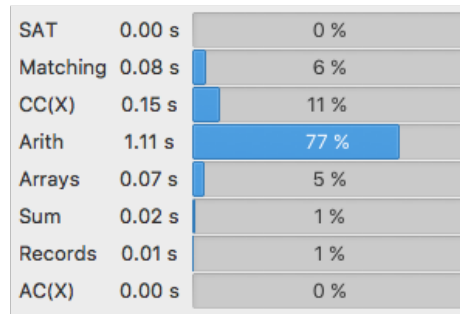| SAT	0.00 s	0 %
Matching	0.08 s	6 %
CC(X)	0.15 s	11 %
Arith	1.11 s	77 %
Arrays	0.07 s	5 %
Sum	0.02 s	1 %
Records	0.01 s	1 %
AC(X)	0.00 s	0 %

Figure 5: Real time profiling information

Figure 5 shows the state of the profiling information after running an example. From the information displayed here, we can see that the problem mostly stresses the theory of arithmetic in the solver. While the solver is running, users can see which part takes the most time and can follow the evolution. For instance, if the time reported for the theory of arrays is too large in proportion and keeps growing, this can indicate that maybe there are some axioms about arrays which are too permissive. Another use case, for developers, is the possibility to identify problems where the solver is stuck in a particular decision procedure (*e.g.*, if only the timer for this theory increases).

4 Syntactic Operations

A lot of the time when trying to use SMT solvers on real world examples (coming from program verification tools for instance), the size of the logical context and the sometimes heavy axiomatizations (that make liberal use of quantifiers at varying degree of alternation) make the problem hard for purely automated tools. However, only a fraction of the actual context is usually necessary to prove the desired goal. Identifying useful information in very large problems can be challenging. When the information provided by the feedback features described in Section 3 allow to identify a potential issue, AltGr-Ergo offers a number of functionalities to perform syntactic manipulations on the context that is shown in textual format. This allows for an iterative (and slightly interactive) approach to SMT solving, where users can experiment and quantify the effect of different manipulations.

4.1 Pruning

Pruning can be performed very easily in AltGr-Ergo by simply double-clicking on the top-level declaration or sub-formula that one wants to deactivate. Reactivating previously deactivated items is also possible by performing the same action.

Most of the time pruning is not dangerous from a soundness point of view. Removing an axiom only under-constrains the original problem which means that any goal proved valid without some top-level

hypothesis is still valid with in the original context. However we also allow users to prune parts of a formula. In this case validity can be affected depending on the polarity of the removed formula.

Consider the original goal:

```
logic P, Q : int → prop
goal g: ∀ x, y: int. Q(x) ∨ P(y) → Q(x) ∧ P(y)
```

which is trivially invalid. Removing `P(y)` on the left side of the implication or on the right side of the implication changes the validity of this goal. In fact removing both turns this goal into a valid one. AltGr-Ergo will allow users to perform these potentially unsafe operations but will notify the user by showing unsound prunings in red. A session that contains unsound pruning operations cannot be saved either. This feature is still useful from an end-user point of view because it allows to attempt proving goals by strengthening hypotheses or weakening the goal itself. For instance if the goal is a conjunction, a user can try to prove only part of the conjunction and gather information from this attempt to help prove the rest.

4.2   Dependency Analysis

AltGr-Ergo maintains dependency information between declarations, definitions and their use. It is possible to remove the declaration of a logical symbol and all top-level declarations that make use of it in a single action. Conversely, reactivating a previously pruned formula or declaration that uses a symbol will also reactivate its declaration and/or definition.

A usage scenario for this feature, is to quickly disable a symbol for which we know the axiomatization is problematic for the solver, then reactivate part of the axiomatization iteratively in the hope that the solver will not be overwhelmed anymore.

4.3   Manual Instances

Quantifiers are notoriously difficult for most SMT solvers. Unfortunately some application domains such as deductive program verification make heavy use of this feature to encode some domain specific functions. For instance Frama-C has built-in axiomatizations for various memory models of C. These are usually very large and complex.

Quantifier instantiation is a heuristic process for SMT solvers in general (although there exists complete techniques for decidable fragments). Alt-Ergo uses *matching modulo equality*. On the other side of the spectrum, interactive theorem provers like Coq or Isabelle require users to perform instantiation (*i.e.* application) entirely manually. This is because in traditional backward reasoning done in theorem provers, only relatively few applications are necessary and a human can figure out which one to do based on the goal, the context and knowledge about the current proof attempt.

AltGr-Ergo gives users the possibility to perform some instances manually. This is useful for example if one has knowledge that a particular goal cannot be solved without using specific instances of a lemma. AltGr-Ergo will ask for terms to use in the instantiation. These can be constants but also other terms from the context. Instances can also be *partial*, which means that we allow that only some of the variables be instantiated. Instances (partial or not) are finally added to the context as hypotheses (they are shown in the top most right corner of the window). All of the other presented actions can be performed on instances.

4.4 Triggers Selection and Modification

As mentioned in section 2, Alt-Ergo computes triggers—*i.e.* patterns or filters used by the matching algorithm to instantiate axioms—in a *heuristic* way. For certain restricted categories of axiomatizations, there exists techniques for computing triggers that make the instantiation mechanism complete [8], however this is not the case in general and coming up with good triggers is a difficult problem.

Because they essentially control how instances are generated, triggers play an important role in proving goals with quantifiers. For example, consider the following axiom, where f is an uninterpreted function from integers to integers.

```
axiom idempotent : ∀ x : int. f(f(x)) = f(x)
```

The trigger f(f(x)) is more restrictive than f(x). This means that only terms *t* that appear in larger terms f(f(*t*)) will be used for instantiating this axiom. Having this trigger will thus make the solver generate *less* instances of this axiom as opposed to the other one possibility, f(x).

There is a balance to be found when coming up with triggers, between restrictive patterns and liberal ones. In some cases, the solver can be overwhelmed by the instances generated if the triggers are not good, it will likely not terminate from a user's point of view. If patterns are too restrictive, or if Alt-Ergo cannot compute appropriate triggers, the affected axiom will likely not be instantiated enough, preventing the solver of discovering potential inconsistencies. In this latter case, Alt-Ergo will answer "I don't know" which is usually unsatisfactory from an end user perspective.

Triggers can be specified by hand in the source problem for Alt-Ergo. The interface AltGr-Ergo goes a step further by allowing triggers to be modified on the fly in the source buffer window. The problem displayed to the user will show the triggers computed by Alt-Ergo itself (it will also warn the user if the heuristic could not come up with appropriate triggers for a quantified formula) as annotations in the source. These can be modified interactively when one figures that the heuristic did not produce the expected results.

Consider now the following, somewhat degenerate, goal.

```
axiom crazy : ∀ x, y : int. x + 1 = y
goal indeed: false
```

Alt-Ergo will not compute any triggers for the axiom. By default, the solver rejects triggers composed of single variables as they are considered too permissive (any term of the appropriate type can be used for instantiation). However in this specific case, we need to instantiate the axiom with any two integers to discover the inconsistency. Figure 6 shows[6] the functionality, right clicking on the trigger displays a contextual menu to add triggers in a pop-up window. They can be entered by the user, and AltGr-Ergo will then parse and type check the piece of text corresponding to the trigger using the same internal functions (exposed) as the ones of Alt-Ergo. In this example we add the multi-trigger [x, y] which allows to conclude.

Triggers can also be deactivated (*i.e.* removed from the axiom) by using the same deactivation mechanism as the one for formulas. With these possibilities, triggers can be modified at will without resorting on complex solver parameters nor relying on heuristics.

---

[6]Goals in the interface are shown in their negated form, as they will be seen by the solver.
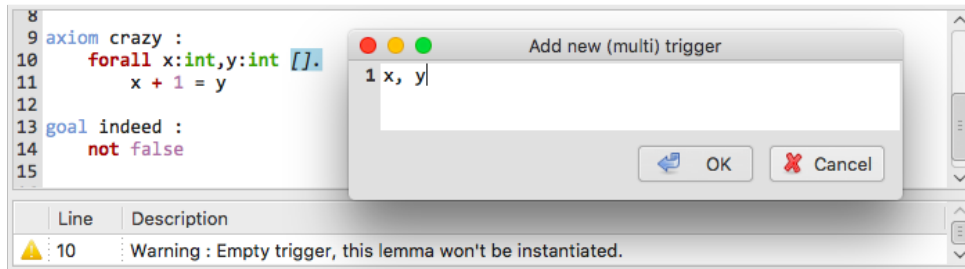
Figure 6: Adding new triggers by hand

4.5   Sessions

Because this interface is designed to provide a slight increase in interactivity for the user, we want all operations to be saved in what we call *sessions* for later *replay*. In particular, operations that manipulate the problem and custom tuning can greatly help the solver in its search. When a user is happy with the state of its modifications—for instance when they allowed to successfully prove the goal—the interface offers the possibility to save the information concerning the session on disk.

This mechanism allows to replay sessions *even if the original problem was modified*. Apart from the list of actions, the only additional information that is saved in the session information is an association table between symbol names for declarations and their identifiers (`id`). Node identifiers in the AST are sequential integers following a depth-first ordering. When a session file is read from the disk, the interface computes offsets for identifiers using this association table in order to figure out the correct corresponding ones for each action in the stack. Of course this is possible only if modifications of the file are relatively minor. For instance, this will work if some axioms were removed or added, or if a modification was performed locally in a declaration (*e.g.*, a formula was changed inside an axiom). Nevertheless, sessions will not survive complete refactorings. If the modifications are too important, the replay will try its best following this offset strategy but can decide to abort if too many actions cannot be replayed.

Another usage of the session replay mechanism is to reuse sessions between problems that share a lot of context. This is useful in a scenario where a user found a satisfying set of modifications and tuning operations on a given problem, and wants to try the same operations on a similar problem (*e.g.*, one where only the goal is different but the context is identical).

5   Implementation

AltGr-Ergo is designed as a new front end for the solver. As such it reuses part of Alt-Ergo's front end and Alt-Ergo's API. The interface is placed at the level of the typed abstract syntax tree (Typed AST on Figure 7) and can manipulate this representation at will. The solver itself communicates various pieces of information to the interface.

The graphical part of the interface is written in GTK, using the OCaml bindings LablGTK and runs in a separate thread. When Alt-Ergo is started in graphical mode, one thread is created for the interface, and one thread is created for every instance of the solver (started with the button "Run"). These threads communicate asynchronously through shared variables, messages or signals depending on the functionality. For example, runs of the solver can be aborted at anytime by clicking the button "Abort", a signal is emitted and caught by the interface and the solver instance.

Most of the work performed by the interface is done on an *annotated* version of the AST.
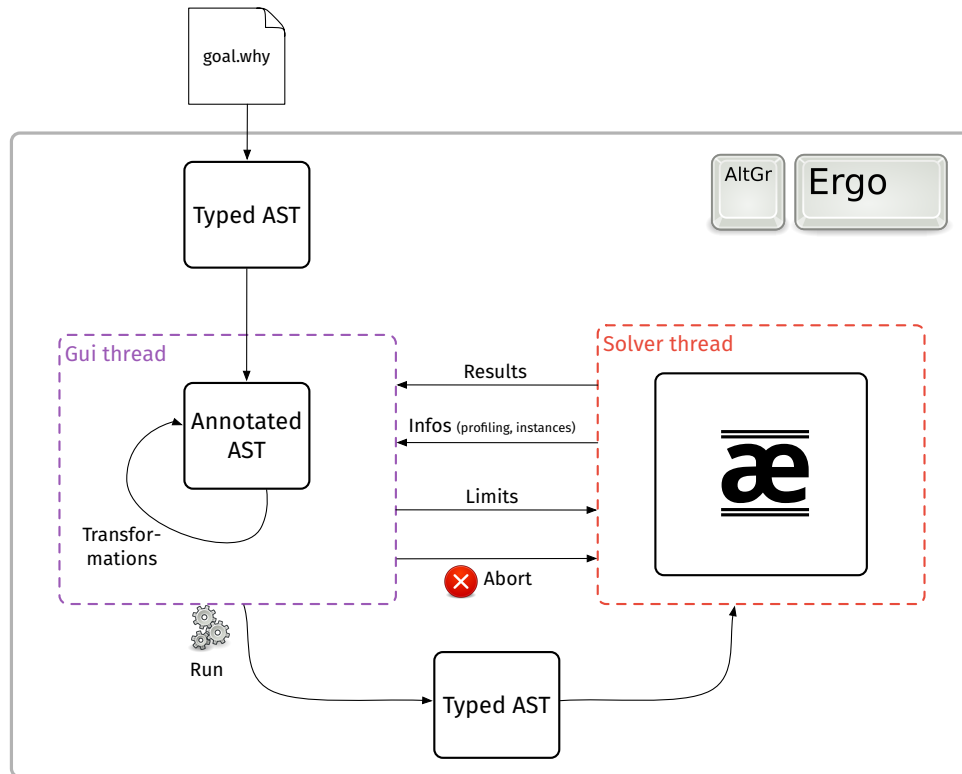
Figure 7: AltGr-Ergo's architecture

5.1   Annotations

An annotated AST node is encoded by the record shown below. It contains the node itself, mutable flags for pruning and polarities, GTK tags, a unique identifier, the GTK buffer in which it is displayed and the line number at which it appears in the source.

```
type 'a annoted = {
  mutable c : 'a;          (* annotated value *)
  mutable pruned : bool;   (* pruned mark *)
  mutable polarity : bool; (* polarity for sub-terms *)
  tag : GText.tag;         (* GTK tag associated *)
  ptag : GText.tag;        (* Another GTK tag for proofs *)
  id : int;                (* Unique identifier *)
  buf : sbuffer;           (* Source buffer *)
  mutable line : int;      (* line numbers can change *)
}
```

Some of these fields are mutable to account for user actions. For instance pruning a sub-term in the source amounts to changing the flag `pruned`. In turn, polarities can be affected by these operations (see Section 4.1).

Tags are used as a way to effectively identify portions of the source code displayed to the user in the buffer. These special tags of type `GText.tag`, allow to efficiently change properties of the display at any moment. For instance they are used to show the user sub-terms of the formulas and their type when

the mouse is hovered over the particular location in the buffer area. Tags can be stacked, this is why we have a second one, `ptag`, which displays unsat cores and remains unaffected by any other action (see Section 3.1). The operations described in Section 4 are all performed on this intermediate representation.

When the user clicks on the button *Run*, annotations are stripped form the AST and the resulting AST is sent to a newly created thread for this specific instance of the solver Alt-Ergo.

5.2   Sessions

Every action performed in the interface is saved in a stack in the current environment. This is the stack that is exported when the session is saved.

```
type action =
  | Prune of int
  | IncorrectPrune of int
  | Unprune of int
  | AddInstance of int * string * string list
  | AddTrigger of int * bool * string
  | LimitLemma of int * string * int
  | UnlimitLemma of int * string

type env =
  { ...
    saved_actions : action Stack.t;
    ... }
```

Actions are parameterized by the identifier of the AST node on which they were performed. Only atomic actions are saved. This is because all other operations can be expressed in terms of these atomic actions. This simple structure allows to replay sessions at a later date and obtain a state which is equivalent to the one that was shown to the user when it was saved on disk.

6   Conclusion and Future Work

We've presented AltGr-Ergo, a graphical front end for the SMT solver Alt-Ergo. It provides real time feedback to users and allows to interactively manipulate problems. We believe such a tool is beneficial both from an end-user and developer point of view, especially for the kind of goals that arise from deductive program verification. We have personally (as the developers of Alt-Ergo) found the interface precious to tune the solver and tackle larger verification problems.

Future directions to improve this tool include turning AltGr-Ergo into a full featured editor for SMT solvers (at the moment the source buffer window cannot be edited directly). There are still some inefficiencies in the interface when one wants to load *very large* files, due to some of the more advanced features, but also inherent to GTK itself. Another future direction for this tool, would be to dissociate the interface from the prover completely with the objective of providing a *web based interface*, which would be more portable, flexible and easy to use. Also, the session mechanism could be further improved to allow for more replay scenarios by integrating some more advanced diff and merge techniques [1].

This work is a first step towards *semi-interactive* theorem provers. One more ambitious goal would be to provide a set of *tactics* and an appropriate language to allow users to do part of their proof manually, or

to finish (also partly manually) proofs that the SMT solver struggle with. This would involve work both at the level of the proof language as well as the level of the user interface.

References

[1] Serge Autexier (2015): *Similarity-Based Diff, Three-Way Diff and Merge*. International Journal of Software & Informatics 9(2). Available at http://www.ijsi.org/ch/reader/view_abstract.aspx?file_no=i217.

[2] John Barnes (2012): *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis, http://www.altran.co.uk, UK.

[3] François Bobot, Sylvain Conchon, Evelyne Contejean & Stéphane Lescuyer (2008): *Implementing Polymorphism in SMT solvers*. In: *SMT '08/BPR '08: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, ACM, New York, NY, USA, pp. 1–5, doi:10.1145/1512464.1512466. Available at http://www.lri.fr/~conchon/publis/conchon-smt08.pdf.

[4] ClearSy System Engineering: *Atelier B User Manual, version 4.0*. Available at http://tools.clearsy.com/wp-content/uploads/sites/8/resources/User_uk.pdf.

[5] Sylvain Conchon, Evelyne Contejean & Mohamed Iguernelala (2012): *Canonized Rewriting and Ground AC Completion Modulo Shostak Theories : Design and Implementation*. Logical Methods in Computer Science 8(3), doi:10.2168/LMCS-8(3:16)2012. Available at http://www.lri.fr/~conchon/publis/conchon-lmcs2012.pdf.

[6] Sylvain Conchon, Evelyne Contejean, Johannes Kanig & Stéphane Lescuyer (2008): *CC(X): Semantic Combination of Congruence Closure with Solvable Theories*. Electronic Notes in Theoretical Computer Science 198(2), pp. 51–69, doi:10.1016/j.entcs.2008.04.080. Available at http://www.lri.fr/~conchon/publis/conchon-entcs08.pdf.

[7] Sylvain Conchon & Mohamed Iguernelala (2016): *Increasing Proofs Automation Rate of Atelier-B Thanks to Alt-Ergo*. In Thierry Lecomte, Ralf Pinger & Alexander Romanovsky, editors: *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - First International Conference, RSSRail 2016, Paris, France, June 28-30, 2016, Proceedings*, Lecture Notes in Computer Science 9707, Springer, pp. 243–253, doi:10.1007/978-3-319-33951-1.

[8] Claire Dross, Sylvain Conchon, Johannes Kanig & Andrei Paskevich (2016): *Adding Decision Procedures to SMT Solvers using Axioms with Triggers*. Journal of Automated Reasoning 56(4), pp. 387–457, doi:10.1007/s10817-015-9352-2.

[9] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 - Where Programs Meet Provers*. In: *ESOP*, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.

[10] Mohamed Iguernelala (2013): *Strengthening the heart of an SMT-solver : Design and implementation of efficient decision procedures. (Renforcement du noyau d'un démonstrateur SMT : Conception et implantation de procédures de décisions efficaces)*. Ph.D. thesis, University of Paris-Sud, Orsay, France. Available at https://tel.archives-ouvertes.fr/tel-00842555.

[11] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles & Boris Yakobowski (2015): *Frama-C: A software analysis perspective*. Formal Aspects of Computing 27(3), pp. 573–609, doi:10.1007/s00165-014-0326-7.

[12] Michał Moskal, Jakub Łopuszański & Joseph R. Kiniry (2008): *E-matching for Fun and Profit*. Electron. Notes Theor. Comput. Sci. 198, pp. 19–35, doi:10.1016/j.entcs.2008.04.078. Available at http://dl.acm.org/citation.cfm?id=1371256.1371287.