Slot machines

an approach to the Strategy Challenge in SMT solving

Stéphane Graham-Lengrand

CNRS - Ecole Polytechnique - SRI International,

SMT-Workshop, 19th July 2015

Disclaimer

This talk

- will not give you hints for winning the next SMT competition
- is not about a new challenger for the SMT competition
- will not address a new theory / a new procedure

is more about software design issues for solvers,

with *experimentation* and *correctness* in mind

Contents

I. The strategy challenge in SMT-solving

"High-performance SMT solvers contain many tightly integrated, hand-crafted heuristic combinations of algorithmic proof methods. While these heuristic combinations tend to be highly tuned for known classes of problems, they may easily perform badly on classes of problems not anticipated by solver developers. This issue is becoming increasingly pressing as SMT solvers begin to gain the attention of practitioners in diverse areas of science and engineering. We present a challenge to the SMT community: to develop methods through which users can exert strategic control over core heuristic aspects of SMT solvers. We present evidence that the adaptation of ideas of strategy prevalent both within the Argonne and LCF theorem proving paradigms can go a long way towards realizing this goal"

[de Moura and Passmore, 2013]

PSYCHE (Proof-Search factorY for Collaborative HEuristics): "A proof-search engine based on sequent calculus with an LCF-style architecture" [Graham-Lengrand, 2013]

Heuristics

"Main" solver mechanisms given by (mostly non-deterministic) rule-based systems

(state transition systems, inference systems, etc)

Example: DPLL

"heuristics play a vital role in high-performance SMT, a role which is all too rarely discussed or championed"

Examples

smaller steps:

- variable selection
- when to learn / forget clauses
- when to restart
- ...

bigger steps:

- pre-processing methods
- combination of search methods
- organising the workflow between different components
- ...

Empowering users

- Clear main motivation:

  use of solvers in contexts (e.g. classes of problems)

  not necessarily anticipated by developers

  How well a tool performs obviously depends on what it is used for

- Critical role of **experimentation** in tuning the heuristics
- Who does the experimentation, and the tuning? Developers? end-users?

- Different levels of users $\Rightarrow$ different levels of control over heuristics?

Tuning heuristical aspects

What is often used is

- parameters (e.g. command-line options, etc)
- portfolios (e.g. [Wintersteiger et al., 2009]),

  where various solvers, or variously tuned versions of a solver, concurrently run
- possibly machine learning techniques

The possibilities, though numerous, are still those anticipated by developers

Strategy challenge proposes to let users **define strategies**

organising the application of more primitive tactics and tacticals

Big-step example,

with a language for defining strategies [de Moura and Passmore, 2013]:

```
simplify ; gaussian ; (modelfinder | smt(apcad(icp)))
```

II. Experimenting without jeopardising correctness

Central to experimentation: correctness

Empowering users should not affect the answers produced by the solver

The main approach to correctness is to **check answers** a posteriori:

- check the model if **sat**
- check the proof object if **unsat** (as those produced by e.g. CVC4, VeriT, Z3, etc)

There may be more convenient approaches for experimentation:

If modifying the solver's behaviour does jeopardise correctness of answers, detecting it from checks may come very late, and may lead to time-consuming de-bugging

The alternative of **verifying** the (whole) solver's code would probably be even more time-consuming and form a major hindrance to experimentation

A third way is to seek a guarantee that answers are **"correct-by-construction"** when designing the strategy definition mechanisms

/ identifying the perimeter of "heuristical experimentation"

Strategy language

A strategy language that authorises big-step strategy definitions such as

```
simplify ; gaussian ; (modelfinder | smt(apcad(icp)))
```

may be sufficient to ensure correctness of answers,

while providing enough expressivity for end-users

But *developers* also need to experiment:

"By introducing a strategy language foundation into our solvers, we [as solver developers] have found our productivity radically enhanced, especially when faced with the goal of solving new classes of problems. The strategy language framework allows us to easily modify and experiment with variations of our solving heuristics. Before we had such strategy language machinery in place, with its principled handling of goals, models and proofs, this type of experimentation with new heuristics was cumbersome and error-prone." [de Moura and Passmore, 2013]

Developers probably want to also tweak smaller-step heuristical aspects

Can we guarantee correctness when experimenting with the **source code**?

The kernel-plugin architecture

Use a modular software architecture that separates

- the code implementing the actual rules of your rule-based system — kernel

  concerns **correctness** of answer
- the code implementing the strategies that determine their application — plugin

  concerns **efficiency** of producing answer

A metaphor: Kernel = a car moving on a road network

Plugin = driver in the car

Common objective: reach a destination

Correctness:

interaction between Kernel and Plugin is organised so that the car stays on the road

cannot claim the destination is reached if it isn't

In other words: trust the car for correctness, hope driver is efficient at driving it

Driver gets into unfamiliar neighbourhood? — Change driver!

No proof-checker needed — . . . if you trust kernel's correctness

LCF style

This can be based on the **LCF** principle now widely used in Interactive theorem proving (e.g. HOL-Light, Isabelle, etc):

Kernel knows of private type **`thm`** for theorems

(constructors of **`thm`** not known outside kernel)

offers API so proof-construction becomes programmable outside kernel

producing inhabitants of **`thm`**

Given inference rule

$$\frac{\text{prem}_1 \quad \ldots \quad \text{prem}_n}{\text{conc}}\ \text{name}$$

kernel offers API *top-down* primitive `name:` **`thm`** `->` $\cdots$ `->` **`thm`** `->` **`thm`**

$\Longrightarrow$ inhabitants of **`thm`** are trusted as proved theorems if kernel is trusted

(regardless of the rest of the code)

Proof-search in LCF

LCF highly programmable, but kernel is of little help for the proof-search per se:

LCF primitives are for proof reconstruction rather than proof-search, usually performed bottom-up, for which tactics and tacticals are used.

A tactic can break a goal into subgoals; the way a proof of the goal can be reconstructed (via the LCF primitives) from those of subgoals, is recorded somewhere (Continuation-Passing-Style can be used)

Tacticals are tactics combinators

Kernel does not organise exploration of search-space, especially backtracking
LCF architecture guarantees *soundness* of answer, not *completeness*
(i.e. "**unsat**" answers are guaranteed correct, not "**sat**" answers)

III. The slot machine approach

The approach in PSYCHE

Kernel knows search-space, which portion has been, or remains to be, explored

(takes branching and backtracking into account)

Plugin drives kernel through search-space (which branch explore first? which depth?)

Kernel says when a proof has been found, or no proof exists Not the plugin

Safety of output

PSYCHE's kernel primitive of the form `machine:` **statement** `->` **output**

inference rule

$$\frac{\text{prem}_1 \quad \ldots \quad \text{prem}_n}{\text{conc}}\ \text{name}$$

is used in that `machine(conc)` triggers recursive calls

`machine(prem_1)`,...,`machine(prem_n)`

What exactly is type **output**?

Kernel = slot machine

Output type of solver:

```
type answer  = Provable of statement*proof
             | NotProvable of statement*model
```

Output (recursive) type of kernel primitive `machine`:

```
type output  = Jackpot of answer
             | InsertCoin of info*(coin->output)
```

The kernel's machine

- either outputs final answer provable or not provable
- or pauses proof-search because a heuristical choice needs to be made:

  for computation to continue, "another coin needs to be inserted in the slot machine";

  depending on coin, search will resume in a certain way.

Division of labour

A *plugin* provides a function `solve` of type: **output->answer**

Top-level call

```
Plugin.solve(Kernel.machine(Parser.parse input))
```

Kernel knows the rules,

applies systematic steps that can be performed wlog and without intelligence

until hits point that requires smart choice to be made by heuristic

At each of those points, plugin instructs kernel how to make the choice

Some choices ("don't know non-determinism") do not preserve provability

(= over-approximations, in SMT terminology), so Kernel records alternative choices

which will have to be tested if over-approximation is **sat**

Kernel realises by itself when search space is exhausted, when problem is **unsat** or **sat**

In fact, *any* function of type **output->answer** forms a valid plugin;

programmability of heuristics has access to the full expressivity of the programming language

Correctness

```
type answer  = private
             Provable of statement*proof
             | NotProvable of statement*model
```

For plugin (of type **output->answer**), **answer** is private to kernel:

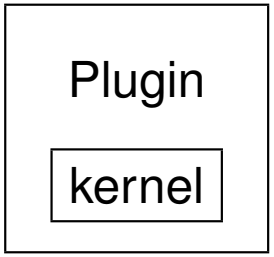
it cannot construct a value of that type,

can only pass on a value provided by (`Kernel.machine`)

= plugin cannot cheat

= no need to understand or verify plugin's code to have a guarantee about the output

In brief,



Plugin in control of workflow,

inserts coins into kernel to make its internal state evolve

until kernel reaches final answer

IV. Examples

DPLL

- **Decide**:

  $\Gamma \| \phi \Rightarrow \Gamma, l^d \| \phi$ where $l \notin \Gamma, l^\perp \notin \Gamma, l \in \text{lit}(\phi)$

- **Fail**:

  $\Gamma \| \phi, C \Rightarrow$ UNSAT if $\Gamma \models \neg C$ and there is no decision literal in Γ

- **Backtrack**:

  $\Gamma_1, l^d, \Gamma_2 \| \phi, C \Rightarrow \Gamma_1, l^\perp \| \phi, C$ if $\Gamma_1, l, \Gamma_2 \models \neg C$ and no decision literal is in Γ_2

- **Unit propagation**:

  $\Gamma \| \phi, C \vee l \Rightarrow \Gamma, l \| \phi, C \vee l$ where $\Gamma \models \neg C, l \notin \Gamma, l^\perp \notin \Gamma$

lit(ϕ) denotes the set of literals that appear / whose negation appear in ϕ

Very small-step heuristics

Branching literal clearly a choice to be made by heuristics

All three other rules: determined by choice of clause

What is considered a basic mechanism in rule-based system A

. . . might appear as a strategy for rule-based system B:

Example: the rules of DPLL can be seen as a specific proof-search strategy for bottom-up proof-search in sequent calculus

. . . while UP, say, is probably considered a basic step by most

Still, considerering it a strategy allows your mechanism for watched literals to take place outside kernel, so that tweaking the mechanism is guaranteed not to affect correctness

To alleviate kernel ("trusted base"), conflict analysis and clause learning can also be taken out of kernel.

One way is to have plugin **memoise** the values returned by the machine

Clause forgetting, restarts are definitely of heuristical nature:

to restart, plugin can record 1st-ever value of plugin-kernel interaction & resume there

PSYCHE in a nutshell

PSYCHE implements the slot machine kind of kernel/plugin interaction

with sequent calculus as inference system

designed as a platform for automated or interactive theorem proving

In version 1.2 for pure propositional logic, kernel was $<$ 500 l.o.c

(but not necessarily restricted to CNF)

Plugin can be as short as 10 l.o.c.

Can be enriched incrementally to compute smarter and smarter strategies

- Great for teaching:

  Kernel directly implements rules seen in the lecture, nothing else

  Incremental sophistication of plugins gives rise to practicals.

  Same testing platform usable at every stage

  Proof objects can be output and displayed in e.g. LaTeX (though quickly too big)

- Now kernel is bigger because it deals with theory-specific decision procedures and unrestricted quantifiers

Theories

Inference systems widely used to describe theory solvers as well

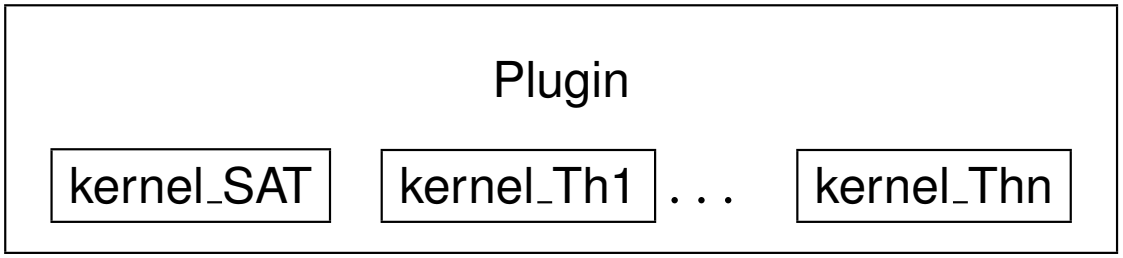
Inference module axiomatised for instance in [Ganzinger et al., 2004] to generically describe theory combination methods (Nelson-Oppen, Shostak, etc)

Again, generally non-deterministic systems.

Heuristics needed for each theory

Even more heuristical aspects in their combination:

- distribute CPU time between theories
- organise the propagation of literals and learnt lemmas
- . . .



Plugin in control of workflow:

inserted coins used to feed other theories' propagated literals and learnt clauses

until all theories agree on model or one of them says **unsat**

V. Limitations and conclusion

Too much freedom?

Slot machines give a control over strategies that is

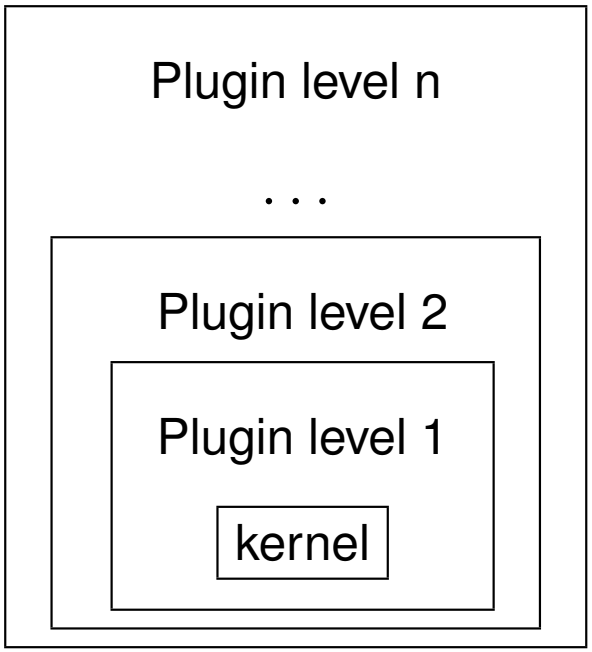
- **powerful**

  access to the full expressivity of the programming language
- **fine-grained**

  may delegate to plugin any little choice that does not affect answer

End-users may not want / know how to harness so much freedom

Different levels of users $\Rightarrow$ different levels of control over heuristics



big step heuristics

small step heuristics

Above certain levels, may want to have specific strategy languages, as in [de Moura and Passmore, 2013]

Efficiency?

Main question about this software design approach: Can it be done efficiently?

Clearly, there will be overheads

Not so much in the going back and forth between components (kernel/plugin)

More problematic are **data-structures** (e.g. terms, literals, sets of literals, etc):

Kernel cannot rely on plugin's data-structures (could jeopardise correctness)

Has to perform its tasks with its own data-structures

. . . that will be agnostic to the used heuristic

Plugin probably needs data-structures appropriate for its "tricks" (e.g. watched literals)

Conversions between representations

or (as currently in PSYCHE) constant maintenance of double-representation of data

clearly incur an overhead

Price to pay for comfort of safe heuristics experimentation

To do

Unable to quantify that overhead yet, because

- Encoding DPLL as proof-search strategy in the sequent calculus is even finer-grained, and incurs an extra overhead
- More importantly, small-step strategies currently implemented in PSYCHE are very naive / illustrative toys.

  For instance for SAT-solving:
  - decision literals taken in whichever order they come
  - no bimp or timp tables,
  - 2-watched literals not implemented on learned clauses
  - etc

  LRA decision procedure also basic, not incremental, etc

  But PSYCHE is a platform where people knowing good and efficient techniques should be able to program them, without worrying about correctness

Thank you!

www.lix.polytechnique.fr/~lengrand/Psyche

References

[de Moura and Passmore, 2013] de Moura, L. M. and Passmore, G. O. (2013). The strategy challenge in SMT solving. In Bonacina, M. P. and Stickel, M. E., editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *LNCS*, pages 15–44. Springer-Verlag.

[Ganzinger et al., 2004] Ganzinger, H., RueB, H., and Shankar, N. (2004). Modularity and refinement in inference systems. Technical Report SRI-CSL-04-02, SRI.

[Graham-Lengrand, 2013] Graham-Lengrand, S. (2013). Psyche: a proof-search engine based on sequent calculus with an LCF-style architecture. In Galmiche, D. and Larchey-Wendling, D., editors, *Proc. of the 22nd Int. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux'13)*, volume 8123 of *LNCS*, pages 149–156. Springer-Verlag.

[Wintersteiger et al., 2009] Wintersteiger, C. M., Hamadi, Y., and de Moura, L. M. (2009). A concurrent portfolio approach to SMT solving. In Bouajjani, A. and Maler, O., editors, *Proc. of the 21st Int. Conf. on Computer Aided Verification (CAV'09)*,

volume 5643 of *LNCS*, pages 715–720. Springer-Verlag.