




SOPRANO

Novel Automatic Solver for Program Analysis

SOPRANO

14/04/2015
D1.1-v1-68-g0ec0970

Combination frameworks

 SOPRANO <i>Novel Automatic Solver for Program Analysis</i>	Confidentiality		Client
	Defense No	Industry No	ANR
	Contractual Yes		Funding ANR
	WorkPackage D2.1-D2.2		
Title Combination frameworks			
Author(s) Sébastien Bardin – CEA François Bobot – CEA Zakaria Chihani – CEA Sylvain Conchon – UPSUD			
Revision D1.1-v1-68-g0ec0970			Number of pages 22
Abstract A solver should be able to address different types and theories in order to be usable on real examples. These theories could be radically different, for example the reals that are uncountable and the booleans that have only two elements. In order to conciliate them, solvers use techniques called <i>combination techniques</i> . This report proposes a survey on existing combination techniques and presents a first formalization of the new combination scheme developed in the SOPRANO solver.			
Keywords automated reasoning, combination techniques, decision procedure, nelson-oppen			
Partners UPSUD CEA		Leader of the Workpackage INRIA	
SOPRANO (ANR-14-CE28-0020) from The National Research Agency (ANR) – © 2016			

Contents

Introduction	5
State of the art	7
2.1 Basics: theories, models and solvers	7
2.2 Nelson-Oppen	8
2.3 Shostak theories	11
2.4 Constraint Programming	12
Popop Domain Framework	13
3.1 Basics	14
3.2 Operations	15
3.3 Development	19
References	19
Bibliography	21

Introduction

A solver should be able to address different types and theories in order to be usable on real examples. These theories could be radically different, for example the reals that are uncountable and the booleans that have only two elements. In order to conciliate them, solvers use techniques called *combination techniques*.

This report is organized as follows, in a first part a survey on existing combination techniques and in a second part a first formalization of the new combination scheme developed in the SOPRANO solver.

State of the art

2.1 Basics: theories, models and solvers

Definitions

A signature Σ is composed by a set Σ^C of constants, a set Σ^F of function symbols, and a set Σ^P of predicate symbols. The logical symbols $\vee, \wedge, \approx, \neg$ they are predefined and are not considered as symbols of any theory. We use the standard notions of (Σ -)term, atom, literal, formula. We use \approx to denote equality's logical symbol. We identify a conjunction of formulas $\phi_1 \wedge \dots \wedge \phi_n$ with the set ϕ_1, \dots, ϕ_n .

If ϕ is a term or a formula, $\text{vars}(\phi)$ denotes the set of variables occurring in ϕ . Similarly, if Φ is a set of terms or a set of formulas, $\text{vars}(\Phi)$ denotes the set of variables occurring in Φ .

For a signature Σ , a Σ -interpretation \mathcal{A} with domain A over a set V of variables is a map which interprets each variable x as an element $x^{\mathcal{A}} \in A$, each constant $c \in \Sigma^C$ as an element $c^{\mathcal{A}} \in A$, each function symbol $f \in \Sigma^F$ of arity n as a function $f^{\mathcal{A}} : A^n \rightarrow A$, and each predicate symbol $P \in \Sigma^P$ of arity n as a subset $P^{\mathcal{A}}$ of A^n . We adopt the convention that calligraphic letters $\mathcal{A}, \mathcal{B}, \dots$ denote interpretations, while the corresponding Roman letters A, B, \dots denote the domains of the interpretations.

For a Σ -term t over V , we denote with $t^{\mathcal{A}}$ the evaluation of t under the interpretation \mathcal{A} . Likewise, for a Σ -formula ϕ over V , we denote with $\phi^{\mathcal{A}}$ the truth-value of ϕ under the interpretation \mathcal{A} . If T is a set of Σ -terms over V , we denote with $T^{\mathcal{A}}$ the set $\{t^{\mathcal{A}} \mid t \in T\}$.

A formula ϕ is satisfiable, if it is true under some interpretation, and unsatisfiable otherwise.

A Σ -structure is a Σ -interpretation over an empty set of variables.

A Σ -theory T is a class of Σ -structure called T -model [9]. A formula ϕ

is T -satisfiable if it is satisfied by some T -model, and it is T -unsatisfiable otherwise. The union of two theories T and S , written $T \cup S$, is a set of $(\Sigma \cup \Omega)$ -interpretation which is in T when restricted to Σ symbols and in S when restricted to Ω symbols.

Example 1. *The theory of arrays is defined by all the interpretations that verify the following axioms:*

$$\begin{aligned} \forall a, x, v. \mathit{select}(\mathit{store}(a, x, v), x) &\approx v \\ \forall a, x, y, v. x \not\approx y &\implies \mathit{select}(\mathit{store}(a, x, v), y) \approx \mathit{select}(a, y) \end{aligned}$$

A solver for a theory T is an algorithm that, given a set of Σ literals, tells if they are T satisfiable or unsatisfiable. A solver is said *complete* if it is correct when it says that ϕ is unsatisfiable. A solver is said *sound* if it is correct when it says that ϕ is satisfiable.

Let S^T a solver of the theory T , and S^S a solver of the theory S , the combination of S^T and S^S consists in creating a solver for $T \cup S$ using S^T and S^S . A trivial problem is that generally the $(\Sigma \cup \Omega)$ -terms are not accepted by both S^T and S^S since they will use unknown symbols. But more importantly even if the terms do not mix, the symbols, and literals are sent to the corresponding solver, the two subset of the problem could be satisfiable but not the problem itself. Indeed the solvers could disagree on the interpretation.

$$f(x) \neq f(y) \wedge x - y = 0$$

Thus the need to design combination techniques.

2.2 Nelson-Oppen

The Nelson-Oppen (NO) technique[15] aims at being able to use efficient decision procedures together. Previous research on automated reasoning focused on specific theories, with few symbols such as theory of integers under $+$ and \leq , the theory of arrays under *select*, *store* ($\mathit{select}(\mathit{store}(a, x, v), x) = v$), the theory of equality with uninterpreted function symbols. Each of these theories' decision procedures works very differently from each other, so the combination technique does not require additional hypothesis on them. So for NO a decision procedure is an algorithm that, given a set of literals, tells if they can be satisfied at the same time or not. The basic NO combination technique will take decision procedures for specific theories and create a decision procedure for the union of these theories. Firstly we will see some formal definitions, a description of the basic combination technique, the limitation of the applicability and, finally, optimization of the decision by requiring more operations from the decision procedure. Secondly we will go through extensions that have been proposed.

2.2.1 Approach

Let Σ and Ω two disjoint signature, let S^T a solver of the theory T , and S^S a solver of the theory S .

The combination method consists of three steps, described below:

1. **Purification step** This phase separates the T and S parts. Every occurrence of a Σ symbol as parameter of a Ω symbol (and conversely) is replaced by a fresh variable, and an equality between this variable and the replaced term is added. At the end the resulting set of literals can be partitioned into the set ϕ_T of Σ -literals and the set ϕ_S of Ω -literals. $\phi_T \wedge \phi_S$ and ϕ are equisatisfiable in $(\Sigma \cup \Omega)$ -literals.
2. **Arrangement step** This phase nondeterministically guesses an equivalence relation E over the set $\text{vars}(\phi_T) \wedge \text{vars}(\phi_S)$ of variables shared by ϕ_T and ϕ_S . From this equivalence relation E the set of literal ϕ_E is generated:

$$\begin{aligned} \phi_E = & \{x \approx y \mid x, y \in \text{vars}(\phi_T) \cup \text{vars}(\phi_S), \text{ and } x =_E y\} \\ & \{x \not\approx y \mid x, y \in \text{vars}(\phi_T) \cup \text{vars}(\phi_S), \text{ and } x \neq_E y\} \end{aligned}$$

3. **Checking step** If $\phi_T \wedge \phi_E$ is T -satisfiable and $\phi_S \wedge \phi_E$ is S -satisfiable, ϕ is satisfiable. Otherwise the check fails.

If, for all the possible equivalence classes of the arrange phase, the check phase fails, ϕ is unsatisfiable.

Soundness and correctness

The technique is complete since for every \mathcal{A} ($T \cup S$)-model an equivalence relation E corresponds and if a subset of a set of literals is unsatisfiable for a subset of the theory, the set of literals is unsatisfiable for all the theory.

The proof of soundness is more interesting and requires that the two theories are stably infinite.

Definition 1. *A Σ -theory T is stably infinite if every quantifier-free Σ -formula ψ is T -satisfiable if and only if it is satisfied by a T -interpretation \mathcal{A} whose domain A is infinite.*

Since the two theories are supposed stably infinite, $\phi_T \wedge \phi_E$ is satisfied by an infinite T -interpretation \mathcal{A} and $\phi_S \wedge \phi_E$ is satisfied by an infinite S -interpretation \mathcal{B} . Since $\text{vars}(\phi_E)$ are the only symbols shared by \mathcal{A} and \mathcal{B} in ϕ_T and ϕ_S , they both satisfy ϕ_E and they have the same cardinality, a $(T \cup S)$ -interpretation \mathcal{C} can be built from \mathcal{A} and \mathcal{B} that satisfies ϕ_T and ϕ_S . So ϕ is satisfiable.

At the end the hypothesis on which original NO apply are that the two theories must be disjoint and stably-infinite. We will see that some refinements of NO are able to lift the stably-infinite part, but none remove the need of disjointness.

Propagations

Additionally the technique allows for one solver to send implied equalities or disequalities to the other solvers. It reduces the number of choices that the combination technique must make and it allows to reach unsatisfiability faster.

Non-convex theories

Non-convex theories are theories that introduce equality disjunctions. Formally let C be a conjunction of literals and e_1, \dots, e_n equalities, a theory is convex if when $\neg(C \implies (e_1 \vee \dots \vee e_n))$ is unsatisfiable then there exists i such that $\neg(C \implies e_i)$ is also unsatisfiable. The theory of linear rational arithmetic is convex and the theory of linear integer arithmetic is non-convex. All the theories with finite domains are non-convex. The non-convex theories often need to do case analysis. With the limited interface given to the solver they have to do it internally. Since case analysis is very time consuming, it is preferable to do it in one place such as in the DPLL(T) framework[16], where the solvers are inside the backtracking engine.

An interface is added that allows the solver to communicate to the framework the disjunction to do. Nelson and Oppen already proposed an extension of their combination framework where theories instead of propagating one equality can propagate a disjunction of equality. The more general split-on-demand[?] allows to send arbitrary clauses.

The cases of non-convex theories with finite domain need a generalization of the stably-infinite hypothesis for being combined by NO.

2.2.2 Refinements

The NO combination technique has been refined in several direction: numbers of pairs to consider, enhanced search procedure of the equivalence relation, more generic theories.

Care graph

The Care graph technique [11] reduces the number of pairs to consider by requiring from the theories to give a graph, called care graph, of what the solver cares about. Only the equality or disequality between the edges of all these graphes will be decided. It allows to directly focus on the potential problems. For example for the solver of uninterpreted functions, given a function f of arity 1, if the terms $f(t_1)$ and $f(t_2)$ are used, t_1 and t_2 are an edge of the care graph. But if they never appear at the same parameter position of the same

symbol, they will not be, since the fact they are equal or disequal does not have any impact on the solver.

Model based combination

Model based combination [5] biases the first choice to try using the partial models of the theories. The idea is to minimize the differences with the current internal model of the theories. For example, if the current assignment given by the simplex of the arithmetic solver sets two terms to the same constant, the first equivalence relation E tries to make these two terms equal.

Soundness hypothesis

Different works tried to loosen the stably infinite hypothesis, since many useful theories do not verify it (*e.g.*, booleans, enumerations). In [21] they used a multi-sorted logic, instead of the mono-sorted presented before, in order to restrict the stably infinite property to the shared sorts. In [18] and completed in [10], this result have been extended to theories of data structures. These theories have the particularity to define their interpretation by using the interpretation of another sort (list of something). These two papers define polite theories, which are theories that can grow the cardinality of their satisfiable interpretation by any amount, and that can create variable witnesses for all the needed elements in a satisfiable interpretation. They prove that any solver can be combined with solver for a polite theory. The care graph technique has also been devised for polite theory in [11]. In [14] they extend the polite notion, to the more general and already existing notion of parametric theories, however it requires not just decision procedures but strong solvers that could reason additionally on constraints about the cardinality of the sorts.

2.3 Shostak theories

Shostak in 1984 [20] proposed a combination algorithm for the theory of equality with uninterpreted symbols and specific equational theories, the so-called *Shostak theories*, for which there exist efficient procedures for, respectively, reducing terms to canonical form (*canonizers*) and turning equations into substitutions (*solvers*). Examples of such theories include the theories of linear arithmetic, pairs, bit-vectors etc.

This method interleaves canonization, equation solving and substitution application in a very tightly coupled way. For instance, to decide the satisfiability of the following conjunction of literals

$$f(x) - x = 0 \wedge f(2x - f(x)) \neq x$$

the solver of linear arithmetic can be used to solve the first equation $f(x) - x = 0$. The resulting substitution $\sigma = \{f(x) \mapsto x\}$ is then applied to the rest of the formula $\sigma(f(2x - f(x))) \neq \sigma(x)$. This yields the literal $f(2x - x) \neq x$ which in

turn is canonized to $f(x) \neq x$. Applying σ again gives the unsatisfiable literal $x \neq x$.

Shostak's method offered an apparently more efficient algorithm, but of restricted scope. What exactly the scope of Shostak's method is has remained unclear for a long time [19, 2, 8, 13]. Furthermore, the method was based on the claim that the disjoint union of two (and therefore any finite number of) Shostak theories is a Shostak theory. However, the validity of this property received minimal serious attention and Shostak himself provided little evidence that this observation was correct. Nevertheless, the Shostak algorithm has influenced the design of several leading tools for automated verification, including PVS [17], SVC [1], and STeP [4]. It is also closely related to the ICS [7] decision procedure.

2.4 Constraint Programming

Constraint programming (CP) started with different initial requirements. The emphasis is more on local constraints than on global ones (like simplex). With simplifications, CP solvers attach to each term one domain (*e.g.*, integer intervals) and schedule constraints that will read the domains of some terms to improve the domains of another (*e.g.*, the constraints $t_1 + t_2 = t_3$ could read $t_2 \in [-1; -4]$ and $t_3 \in [1; 5]$ and deduce that $t_1 \in [-3; 4]$). If at that time $t_1 \in [0; 10]$, then its domain is updated with $[0; 4]$). At some point, for example when no constraint could improve the domains of any terms, a decision is made, for example the domain of a term is split and one side is tried.

If there are no more choices to be made, every variable has an interpretation, and since the propagation at least propagates constants, the problem is satisfiable. On the other hand if the domain of a term becomes empty, the sequence of choices is impossible and the engine backtracks to the last choice made. If there is no last choice the problem is unsatisfiable. The difference with decision procedures is that the satisfiable interpretation is explicitly given and that all the constraints agree with it.

It is not clear that CP uses a combination framework, since everything seems to be one algorithm. However if we take each constraint separately we could say that the combination framework of CP consists in the explicit sharing of the model constructed by each constraint, in a way similar but a lot more eager than model base combination 2.2.2. The equivalence relation doesn't have to be guessed since the value of all the terms is already guessed. CP handles very easily non-convex theories since it can naturally decide on the domains of terms until obtaining a singleton.

At the end we can distinguish the CP and NO techniques (Shostak being in the middle) by how the communication between the theories is done; CP would be like shared memory (efficient, liberal, hard to do right) and NO would be like message passing (safer, cleaner, restricted).

Popop Domain Framework

The design of the POPOP solver is part of the SOPRANO project. One of the goals of the SOPRANO project is to allow easy addition of solvers of new theories.

As in CP and DPLL/NO, the prover refines a partial interpretation until all the terms are interpreted and verifies the constraint or that a constraint is broken. In SAT the refinement takes the form of setting boolean values to boolean variables. In CP the refinement takes the form of domain splitting or labeling (set a domain to a singleton). In POPOP these choices are abstracted into a notion of possible refinements that we will call the *choices*. For each choice the partial model can be refined in different ways. The fundamental requirement for correctness is that once all the choices have been made the partial interpretation is a full interpretation and that we can check correctly if all the constraints are verified. It is a difference with traditional CP where the propagation must always be correct, here they only must be correct on constants. When a partial interpretation is found to not verify the constraint, information is gathered on which choices were wrong. This information is kept as a new constraint. The constraint must be implied by the initial constraint, it should be verified by every interpretation that satisfies the original problem. It is Conflict Driven Clause Learning (CDCL)[22] except that the literals can be anything that can be interpreted, it doesn't have to be a term of the input language. Finally as in CDCL, it is possible not only to learn to exclude those wrongly made choices, but also others impossible choices with the same reason.

This setting is similar to MC-Sat[6, 12] except that MC-Sat limits the choices of setting variables to constants, and they use the result of propagation only for checking the emptiness of the domain, not for new propagations (no transitive propagation). It is the contrary of what CP does. In POPOP we want to have the possibility to experiment with all the possible spectrum of quantity of propagation, as few as in SAT/MC-SAT or as much as in CP.

Adding a new solver is as simple as in the CP framework, for the refinement

phase (also called propagation phase), since a full interpretation is built at the end. The interface that the solver can interact with is more precise than in CP in a way similar to NO. Moreover contrary to the traditional CP, conflicts are computed as in CDCL solvers. We will devise these techniques and their integration in SOPRANO in report D2.3. So for now we will just say that from a conflict some information can be gathered to indicate which choices are impossible.

In the next sections we will dive in the technicalities of POPOP. Popop tries to be quite general because it should be a play-ground for testing different trade-offs. First will see the basic notions, then the interface between the engine and the solvers, finally solvers for different theories.

3.1 Basics

The engine does not manipulate terms but only equivalence class cl . An equivalence class can be obtained from a term but from an equivalence class one can not obtain the set of terms it represents. To each equivalence class domains dom are associated.

The terms are not directly converted to cl . Usual terms are not easy data-structure to work with. Depending on the theory, particular ones are more suited. For example, for linear arithmetic, a mapping from cl to rational constants is simpler than a tree of additions and multiplications. Moreover the part of the terms that is not part of linear arithmetic is abstracted using its equivalence class, in a way similar to the purification phase 1 of NO. These particular data-structures are called semantical values since they have a similar role to that in Shostak 2.3. To each semantical value is associated a cl in an injective way. Fresh equivalence class could also be created for example to describe top-level existential variables.

New domains and new kind of semantical values can be defined. One can also add daemons, that are pieces of code that would be executed when some events happen. The engine does not directly recognize the notion of theory, it is only a high-level notion. We consider that a POPOP's theory is a set of domains, kinds of semantical values and daemons used together.

The terms are converted to cl directly after parsing in a bottom-up pass, using semantical values along the way.

Example 2. *The term $a \approx a + f(a, a + 5) + a + 3$ is the equivalence class cl_8*

with

$$\begin{aligned} \text{cl}_8 &\leftarrow \{\text{cl}_0, \text{cl}_7\}_{eq} \\ \text{cl}_7 &\leftarrow \{3, \text{cl}_0 \mapsto 2, \text{cl}_6 \mapsto 1\}_{arith} \\ \text{cl}_6 &\leftarrow \text{App}(\text{cl}_1, \text{cl}_5) \\ \text{cl}_5 &\leftarrow \text{App}(\text{cl}_0, \text{cl}_4) \\ \text{cl}_4 &\leftarrow \{5, \text{cl}_0 \mapsto 1\}_{arith} \end{aligned}$$

$\{\text{cl}_0, \dots, \text{cl}_n\}_{eq}$ has the interpretation of a disjunction of equality $\bigvee_{i \neq j} \text{cl}_i^A = \text{cl}_j^A$. The set is useful for representing the **distinct** operator. $\text{distinct}(\text{cl}_0, \dots, \text{cl}_n)$ is represented concisely by $\neg\{\text{cl}_0, \dots, \text{cl}_n\}_{eq}$.

$\{c, \text{cl}_0 \mapsto c_0, \dots, \text{cl}_n \mapsto c_n\}_{arith}$ has the interpretation $c^A + c_0^A * \text{cl}_0^A + \dots + c_n^A * \text{cl}_n^A$ and we call them arithmetic polynomials, $\mathcal{P} = \mathcal{Q} \times (\mathcal{Cl} \xrightarrow{\text{finite}} \mathcal{Q})$ ($\mathcal{Cl} \xrightarrow{\text{finite}} \mathcal{Q}$ is the set of mappings p from \mathcal{Cl} to \mathcal{Q} with finitely many cl that have $p(\text{cl}) \neq 0$).

3.2 Operations

The POPOP's engine provides a number of operations that can be used when implementing theories, and some others that theories must provide so that the engine interacts with them.

Theories interact with the engine through the following operations:

- **get_dom**: get the current domain associated to an equivalence class
- **get_repr**: get the representative of an equivalence class
- **set_dom**: modify a domain of a cl. Done immediately, *i.e.*, not queued
- **merge_cl**: ask to merge two cl
- **set_sem**: ask to merge the cl of a semantical value and another cl
- attach a direct daemon to an event with some additional data
- attach a delayed daemon to an event with some additional data
- **register_choice**: register the need for a choice
- **conflict**: raise that a conflict has been found

Possible events are the following :

- **event_{dom}**: when a specified domain of a specified cl changes

- `eventreg`: when a specified `cl` is registered
- `eventsem`: when the `cl` of any semantical value of a specified kind is registered
- `eventrepr`: when a class is no longer the representative of its class

Since from a semantical value one can obtain its associated `cl`, `setsem` seems redundant with `mergecl`. In fact `setsem` has a specific behavior when the `cl` of the semantical value is not registered; it make sure this `cl` is not the new representative. That enforces that if one does not create a new semantical class, one just adds a new semantical value to an existing class, the number of representative is bounded, and so termination can be proved.

The operation `register_choice` allows a theory to indicate that it needs to make a choice. It is similar to domain splitting or labeling done in CP, in the same way that there are heuristics for prioritizing these choices in CP, priorities can be given to these choices. It is also similar to split-on-demand[3] from the SMT community, where a theory can send a disjunctive lemma to the SAT solver. However, the theories do not have to build a boolean formula, the choice is more abstract, and the boolean theory of POPOP uses this mechanism to some of its boolean choices.

The engine interacts with each domain \mathcal{D} through the following operations:

- `merged`: tell if two values of the domain are merged
- `merge`: ask to merge the domain \mathcal{D} of two given classes
- `defdom`: a default element for the domain, used when a domain have not yet been associated to an equivalence class
- `do_choice`: ask if a previously registered choice is still needed, and if it is the case to modify the state of the engine accordingly (for example with `setdom`)

Before doing the operation `do_choice`, the engine sets a backtracking point by saving its state so that it could come back to this point during the conflict analysis phase started by `conflict`.

The engine does not apply the modification immediately otherwise daemons would be executed during execution of other daemons which makes it very hard to write them, to reason about them and to enforce invariants. Therefore, all modifications are delayed, except `setdom` and `setsem`. Moreover they are applied with the following priorities, the highest priority first:

1. `setdom`: modify a domain of a `cl`
2. `setsem`: ask to merge a `cl` and the `cl` of a semantical value

3. `direct_daemon`: run a direct daemon for which an event occurred
4. `merge_dom` (internal): merge a domain of two cl
5. `merge_cl_end` (internal): check that the domains of two cl are merged, fail if it is not the case, otherwise make one the representative of the other
6. `merge_cl`: start merging two classes
7. `delayed_daemon`: run delayed daemon for which an event occurred

Further priority can be applied for the order of execution of the delayed daemons. On the contrary the direct daemons are executed in a fixed first-in-first-out way.

The direct daemons are useful for restoring invariants, since they have a high priority and fixed execution order.

3.2.1 Booleans

The boolean theory has no specific capabilities. It defines $\mathcal{S}_{\text{boo1}}$ which represents a negated or non-negated conjunction of negated or non-negated cl: $\mathcal{S}_{\text{boo1}} = \{\top, \perp\} \times (\mathcal{C}l \mapsto^{\text{finite}} \{\top, \perp\})$. Let an interpretation \mathcal{A} and $(n, m) \in \mathcal{S}_{\text{boo1}}$, its interpretation is:

$$(n, m)^{\mathcal{A}} = n \otimes \bigvee_{cl \in m} m(cl) \otimes cl^{\mathcal{A}}$$

with \otimes the xor function (we recall that $\perp \otimes b = b$ and $\top \otimes b = \neg b$)

We are not using conjunctive normal form because it is needed neither for propagation nor for learning, moreover it simplifies the handling of boolean formulas inside terms (`ite`). The negation is not a separate semantical value because it reduces the number of classes to create. Otherwise on usual problems, there are two classes per boolean variable instead of one : one positive and one negative.

The boolean domain is the usual boolean lattice $\mathcal{D}_{\text{boo1}} = \{\{\top\}, \{\perp\}, \{\perp, \top\}\}$, except the bottom value $\{\}$ is not present because the conflict is diagnosed (and `conflict` used) before updating the domain. The default value `def_dom` is $\{\perp, \top\}$ the top of the lattice. The test `merged` is the set equality. The function `merge` is the set intersection of the domain of the two classes to merge, except if they are disjoint where it uses `conflict` to indicate a conflict.

For each boolean variables cl, a delayed daemon waits on `event_reg` that this cl is registered and in that case register `register_choice` the choice to set it on $\{\top\}$ or $\{\perp\}$.

A delayed daemon waits with `event_sem` for the registration of a class cl of the semantical value of kind $\mathcal{S}_{\text{boo1}}$, when that happens it attaches the event

`eventdom` for `cl` and all the classes contained¹ in the semantical value to the propagation daemon with the semantical value as data.

The propagation daemon, is a delayed daemon that propagates forward and backward information. When the engine wakes up the daemon with a semantical value $(n, m)\mathcal{S}_{\text{bool}}$ associated to the class `cl`:

- if for a class $cl' \in m$, `get_dom(cl) \otimes m(cl) = { \top }` using `set_dom` the domain $\mathcal{D}_{\text{bool}}$ of `cl` is set to $\top \otimes n$ (forward propagation).
- if `get_dom(cl) = { b }` and $b \otimes n = \perp$, for every $cl' \in m$, using `set_dom` the domain $\mathcal{D}_{\text{bool}}$ of cl' is set to $\perp \otimes m(cl')$ (backward propagation).
- if for all but one cl_1 with $m'(cl_1) = n$ the $cl' \in m$, `get_dom(cl') = { \perp }`, then cl_1 is merged with `cl` using `merge_cl` (forward and backward propagation).

If the booleans are combined with another theory, instead of using `set_dom` this propagator merges the class using `merge_cl` to one of two predefined, by the theory, class being true and false. In that way the booleans propagate equalities.

3.2.2 Equalities

The propagation of equalities is straightforward, a daemon waits for each equalities if its boolean values is known or if the terms are equals. The only particular point is that a domain is used to represent that classes must be distinct, it uses the usual tagging technique: each time terms are disequal (the result of the equality is false), we add to their domain a fresh integer. During merge the domain of the two classes are checked to have an empty intersection. During propagation if all the terms have the same tag in their domain, the equality is known to be false.

Even if the equalities are boolean literal, it is not always needed to decide on it. For example if the arguments are booleans we know that at some point it will be decided if they are true or false. So since the input languages of POPOP are sorted, we can distinguish equalities where we will decide on them (eg. uninterpreted sort) and the others (eg. booleans).

3.2.3 Linear rational arithmetic

One can implement Shostak theories, using domains and direct daemons. For example with the theory of linear rational arithmetic equalities. We reuse arithmetic polynomials \mathcal{P} from example 2, and we define a new kind of semantic values $\mathcal{S}_{\text{arith}} = \mathcal{P}$ for arithmetic. The equivalence classes associated

¹In the implementation an algorithm similar to two watched literals allows to wait on less classes, wait until all but one domain class is known.

to arithmetic terms are computed as in example 2 using $\mathcal{S}_{\text{arith}}$. We define a new domain called $\mathcal{D}_{\text{arith}} = \mathcal{P} \cup \{I\}$ which holds the normalized arithmetic polynomial, I is used when the normalized arithmetic polynomial is its owned representative. By overloading the notation we define $\mathcal{D}_{\text{arith}} : \mathcal{Cl} \mapsto \mathcal{P}$ with:

$$\mathcal{D}_{\text{arith}}(\text{cl}) = \begin{cases} \text{get_repr}(\text{cl}) & \text{if } \text{get_dom}(\text{cl}) = I \\ \text{get_dom}(\text{cl}) & \text{otherwise} \end{cases}$$

We define $\text{norm} : \mathcal{P} \mapsto \mathcal{P}$ that substitutes each cl in the polynomial by $\mathcal{D}_{\text{arith}}(\text{cl})$.

The operations on $\mathcal{D}_{\text{arith}}$ and the following direct daemons are defined:

- the **merged** function is the mathematical equality on $\mathcal{P} \cup I$
- a direct daemon waits on $\text{event}_{\text{sem}}$ for $\mathcal{S}_{\text{arith}}$ and , let $p \in \mathcal{S}_{\text{arith}}$ be the semantic value and cl its equivalence class. The domain $\mathcal{D}_{\text{arith}}$ of cl is set to $\text{norm } p$. This invariant is thus enforced during merge.
- the **merge** function, is here similar to the solving function of Shostak 2.3. For two classes to merge cl_1 and cl_2 , the two arguments given to the solve function are $\mathcal{D}_{\text{arith}}(\text{cl}_1)$ and $\mathcal{D}_{\text{arith}}(\text{cl}_2)$. If no substitution exists the conflict is reported using **conflict**. Otherwise the substitution $\text{cl}_s \mapsto p_s$ is recorded by setting the domain $\mathcal{D}_{\text{bool}}$ of cl_s to p_s (cl_s had before the domain I because of the invariant), except if the polynomial is reduced to a class $p_s = \text{cl}_p$, in which case **merge_cl** is call on cl_s and cl_p . The substitution will be applied to the domain of cl_1 and cl_2 by the direct daemon.
- For every equality on rationals a choice is added.

3.3 Development

The refinement/propagation part of the engine is now stable and doesn't change anymore. No special cases have been needed for the currently defined theory (for example for boolean like in SAT or uninterpreted functions like in Shostak). Solver for theories are added one by one, and each time tests are added for new features or when bugs are found. Currently there is 160 tests that are run in default mode and with 9 differents seeds. By default POPOP is completely deterministic, but an option allows to randomize some parts: choice of the representative, choice of the pivot, order of the argument of conjunctions and disjunctions. A test could succeed in the default setting but fail when some randomness is added.

The infrastructure for the learning is present since the beginning but it often changes because there is a lot to invent. It will be described in D2.3.

Bibliography

- [1] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 1996.
- [2] C. Barrett, D. Dill, and A. Stump. A generalization of Shostak’s method for combining decision procedures. In *Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FRODOS)*, volume 2309 of *LNAI*, pages 132–147, Santa Margherita Ligure, Italy, 2002. Springer.
- [3] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. *Splitting on Demand in SAT Modulo Theories*, pages 512–526. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
- [4] N. Bjørner et al. STeP: Deductive-algorithmic verification of reactive and real-time systems. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 1996.
- [5] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electr. Notes Theor. Comput. Sci.*, 198(2):37–49, 2008.
- [6] Leonardo de Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In *VMCAI 2013*.
- [7] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. ICS: Integrated Canonization and Solving (Tool presentation). In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV’2001*, volume 2102, pages 246–249, 2001.
- [8] H. Ganzinger. Shostak light. In *Proceedings of the 18th International Conference on Automated Deduction (CADE)*, volume 2392 of *LNAI*, pages 332–347, Copenhagen, Denmark, 2002. Springer.
- [9] Harald Ganzinger. *Shostak Light*, pages 332–346. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
- [10] Dejan Jovanović and Clark Barrett. Polite theories revisited. In Christian G. Fermüller and Andrei Voronkov, editors, *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR ’10)*, volume 6397 of *Lecture Notes in Computer Science*, pages 402–416. Springer, October 2010. Yogyakarta, Indonesia.
- [11] Dejan Jovanović and Clark Barrett. *Sharing Is Caring: Combination of Theories*, pages 195–210. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

- [12] Dejan Jovanovic, Clark Barrett, and Leonardo De Moura. The design and implementation of the model constructing satisfiability calculus. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 173–180. IEEE, 2013.
- [13] D. Kapur. A rewrite rule based framework for combining decision procedures. In *Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FROCOS)*, volume 2309 of *LNAI*, pages 87–103, Santa Margherita Ligure, Italy, 2002. Springer.
- [14] Sava Krstić, Amit Goel, Jim Grundy, and Cesare Tinelli. Combined satisfiability modulo parametric theories. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Braga, Portugal)*, volume 4424 of *Lecture Notes in Computer Science*, pages 618–631. Springer, 2007.
- [15] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.
- [16] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, November 2006.
- [17] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.
- [18] Silvio Ranise, Christophe Ringeissen, and Calogero G. Zarba. *Combining Data Structures with Nonstably Infinite Theories Using Many-Sorted Logic*, pages 48–64. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
- [19] N. Shankar and H. Rueß. Combining Shostak theories. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications (RTA)*, volume 2378 of *LNCS*, pages 1–19, Copenhagen, Denmark, 2002. Springer.
- [20] R. E. Shostak. Deciding combinations of theories. *JACM*, 31(1):1–12, 1984.
- [21] Cesare Tinelli and Calogero G. Zarba. *Combining Decision Procedures for Sorted Theories*, pages 641–653. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
- [22] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.