A Prototype Implementation of a Distributed Satisfiability Modulo Theories Solver in the ToolBus Framework

David Déharbe[1], Silvio Ranise[2] and Jorgiano Vidal[3]

[1]UFRN/DIMAp
Campus Universitário, Lagoa Nova
59072-970 Natal, RN, BRAZIL
david@dimap.ufrn.br

[2]LORIA & INRIA-Lorraine
615, rue du Jardin Botanique
BP 101
54602 Villers-les-Nancy, FRANCE
Silvio.Ranise@loria.fr

[3]CEFET-RN
Av. Sen. Salgado Filho, 1559
Tirol
59015-000 Natal-RN, BRAZIL
jorgiano@cefetrn.br

Abstract

An increasing number of verification tools (e.g., software model-checkers) require the use of Satisfiability Modulo Theories (SMT) solvers to implement the back-ends for the automatic analysis of specifications and properties. The most prominent approach to build SMT solvers consists in integrating an efficient Boolean solver with decision procedures capable of checking the satisfiability of sets of ground literals in selected theories. Although the problem of checking the satisfiability of arbitrary Boolean combinations of atoms modulo a background theory is NP-hard, there is a strong demand for high-performance SMT-solvers.

*In this paper, we describe the design and prototype implementation of—to the best of our knowledge—the first distributed SMT solver. The emphasis is on providing ways to reduce the implementation effort and to make the system easily extensible. This is achieved in two ways: (*a*) we re-use as much as possible the code of an available sequential SMT solver and (*b*) we adopt the* TOOLBUS *architecture for rapid prototyping. The behavior of the distributed SMT solver was tested on a set of problems which are representative of those generated by software verification techniques. The experiments show the possibility to obtain super-linear speed-ups of the distributed SMT solver with respect to its sequential version.*

Keywords: Satisfiability Modulo Theories, distributed computing, BDDs, haRVey.

1. INTRODUCTION

An increasing number of verification tools (e.g., software model-checkers [10, 2]) require the use of Satisfiability Modulo Theories (SMT) solvers [26] (in first-order logic) to implement the back-ends for the automatic analysis of specifications and properties. This is so because verification problems require to solve satisfiability prob-

lems (e.g., checking if an abstract trace yields a spurious concrete trace can be reduced to a satisfiability problem modulo the theory of the data structures manipulated by the program). So, the availability of efficient SMT solvers becomes a crucial pre-requisite for automating the various verification tasks. To make the situation more complex, most verification problems involve several theories (e.g., programs manipulate composite data structures such as arrays and integers for their indexes), so that methods to combine theories are also required.

There are two prominent approaches to build SMT solvers: *eager* and *lazy*. The former (see, e.g., [8]) is based on *ad-hoc* translations that convert an input formula (and relevant consequences of the background theory) into an equisatisfiable Boolean formula. The approach applies in principle to all theories whose ground satisfiability problem is decidable, possibly at the cost of an exponential blow-up in the translation. The approach is appealing because SAT solvers today are able to quickly process extremely large formulas. The implementation effort is relatively small for being limited to the translator—after that, one can use any off-the-shelf SAT solver. The main disadvantage of the eager approach is that it does not scale up because of the exponential blow-up of the eager translation and the difficulty of combining encodings for different theories.

The lazy approach (see, e.g., [3, 21, 15, 1, 23]), currently the most popular and most successful in terms of run-time performance, consists in building *ad-hoc* procedures to solve the satisfiability problem in a given background theory. The lure of these specialized procedures is that one can use for them whatever specialized algorithms and data structures are best for the background theory under consideration, which typically leads to better performance. The main disadvantage of this approach is that one has to write an entire solver for each new theory of interest. The standard way to overcome this problem is to reduce a theory solver to its essence by separating generic Boolean reasoning from theory reasoning. The common practice is to write theory solvers just for sets of ground literals (i.e. atomic formulas and their negation). These simple procedures are then integrated with an efficient Boolean solver, allowing the resulting system to accept arbitrary Boolean combinations of ground literals. The key idea is to regard the ground first-order atoms of the input formula as Boolean variables and use the Boolean solver to enumerate all its satisfying assignments. As one of these assignments can be seen as a set of ground first-order literals, the decision procedure for the background theory is used to establish its satisfiability. If one assignment is found satisfiable, we are entitled to conclude the satisfiability of the input formula. Otherwise, if all assignments are unsatisfiable, we conclude the unsatisfiability of the formula.

In the *lazy* approach, reasoning modules for a background theory obtained as the union of several simpler theories are modularly built by writing procedures for each component theory and then use the solvers cooperatively via well-known combination schemas (see, e.g., [25] for an overview). More recently, a new schema—called Delayed Theory Combination, DTC—combining the procedures directly with the Boolean solver has been put forward and shown more efficient than SMT solvers based on the classic combination schemas [6].

The problem of checking the satisfiability of arbitrary Boolean combinations of atoms modulo a background theory is NP-hard as it subsumes the problem of checking the satisfiability of Boolean formulas. In spite of the computational complexity, there is a strong demand for high-performance SMT-solvers to make a range of verification techniques viable in practice. In this paper, we describe the design and prototype implementation of—to the best of our knowledge—the first distributed SMT solver based on the lazy approach. The emphasis is on providing ways to reduce the implementation effort and to make the system easily extensible. This is achieved in two ways. First, we re-use as much as possible the code of an available sequential SMT solver. We choose haRVey [15] as the sequential SMT solver since the first two authors are two of its main developers and hence we were more familiar with its design and implementation structure. Second, we adopt the ToolBus [4] architecture in order to obtain a robust implementation of a distributed SMT solver in a short time. This choice is justified by the fact that the ToolBus allows one to write wrappers around a piece of available code and turn it into a module that can be executed concurrently in a distributed environment by exchanging messages according to a protocol that can be easily specified by means of a suitable script language. To show the flexibility of the proposed distributed architecture, we show how a distributed version of DTC can be obtained without too much implementation effort.

After the implementation of the distributed version of haRVey was completed, its behavior was tested on a set of problems which are representative of those generated by software verification techniques, where the sequential version of the solver was already successfully used (see [15, 11]). The main contribution of this work is to show the possibility to obtain super-linear speed-ups for the distributed version of the SMT solver with respect to its sequential version. This is possible by exploiting the following simple observation, which can be generalized to many situations in distributed computing: if we consider $n \geq 2$ Boolean assignments at the same time and invoke n instances of the available procedure on their first-order counterparts, we can hope to significantly reduce the overall running time. As it is well-known in

distributed computing, to make this observation practical, special care must be put into choosing a suitable number n of instances of the decision procedures. Another interesting question is whether there is an "optimal" way to choose the n different assignments to be solved in parallel. Our experiments suggest that the best heuristic in this respect consists of randomly selecting the n Boolean assignments. As we will see, this is the only way to obtain the aforementioned super-linear speed-up.

This article extends a paper previously published in the proceedings of the VIII Brazilian Symposium on Formal Methods [17]. It provides additional technical details on the TOOLBUS framework and gives more information on the algorithm and experimental results. Moreover Section 3.4 is completely original and describes a possible evolution of the distributed SMT-solver to handle new advancements in verification, such as delayed theory combination [6].

Plan of the paper. Section 2 provides the background notion on the TOOLBUS (Section 2.1) and the lazy approach to SMT solving (Section 2.2). Section 3 explains how a distributed SMT solver can be derived from its sequential version (Section 3.1 describes the architecture, Section 3.2 the protocol, and Section 3.3 the internal workings of the various concurrently executing modules) as well as the distributed version of DTC (Section 3.4). Section 4 reports on the experiments. Finally, Section 5 concludes and sketches our future directions of research.

Results presented in this paper have been previously published in [17]. This paper adds on the proceedings version [17] by giving a more detailed presentation of the TOOLBUS framework for the implementation of heterogeneous, distributed systems, in addition to fixing several small mistakes and improved writing. A more fundamental contribution found in this paper over [17] is the inclusion of the treatment of DTC in the already existing framework. This is a strong indicator of the flexibility of the proposed approach to incorporate further technical improvements in the implementation of sequential SMT solvers into the proposed distributed algorithm.

2. BACKGROUND

There are two key ingredients to our implementation of a distributed SMT solver: the TOOLBUS [13] and haRVey [15]. The former allows us to build a prototype and experiment with concrete benefits of using the TOOLBUS; the latter is our implementation of a lazy (and sequential) SMT solver based on the combination of a Boolean solver and satisfiability procedures for some theories, such as equality, lists, arrays, and their combination. We provide some background notions both on the TOOLBUS and lazy SMT-solving to make the paper self-contained.

2.1. THE TOOLBUS

The construction of (heterogeneous) distributed systems is a challenging task, both from a design and implementation points of view. The TOOLBUS [13] provides an elegant solution to implement robust distributed and heterogeneous systems, using a variation of the Algebra of Communication Processes [5] as a script language to describe the protocol between the different components and a uniform, generic, data type (called ATerms, see below for more details). The TOOLBUS framework has two entity classes: *processes* and *tools*. The former are responsible for coordinating actions and synchronizing communications throughout the system, while the latter are the components of the systems which are ultimately responsible for the actual computational work and may be written in a number of different programming languages. All data exchanges are performed using ATerms [29], a term-like data type that features maximal sharing of sub-terms, resulting in a compact representation of symbolic expressions and efficient comparison operators.

The TOOLBUS encourages a methodology whereby programmers write a TOOLBUS script describing the intended interaction protocol between *tools*. Scripts are then directly executable by the TOOLBUS interpreter. Moreover, the TOOLBUS suite provides utilities to automatically generate the interfaces that each tool has to implement to participate in the protocol. Currently, there is support for two programming languages (JAVA and C) and several adapters are available for Perl, ASF+SDF, UNIX scripts, etc.

A TOOLBUS script defines an interaction protocol between different tools by means of a composition of processes. We will adopt the term *tool bus* to denote an instance of such a protocol.

The TOOLBUS scripting language provides the classic process algebra constructs: + for choice, . for sequential composition, $\|$ for parallel composition, * for repetition, if then [else] fi for guarded command, and delta to represent deadlock. The operator create dynamically creates process instances; finally, execute and snd-terminate spawns and aborts the execution of a tool instance, respectively.

The execution of a tool can be dispatched from within the tool bus, with the command execute or it can be started externally by issuing a connection request to the tool bus, that may accept such connection requests with the rec-connect command.

Once connected to the tool bus, tools do not communicate directly. Instead, communication channels are established between tools and processes, or between processes. The communication between processes can be synchronous, using matching send-msg and rec-msg commands, or asynchronous, with the send-note broadcasting command, which can be received using the rec-

```
01  process CALC is
02      let Tid: calc, E: str, V: term
03      in
04          execute(calc, Tid?) .
05          (   rec-msg(compute, E?) .
06              snd-eval(Tid, expr(E)) .
07              rec-value(Tid, val(V?)) .
08              send-msg(result, E, V) .
09              send-note(result(E, V))
10          ) * delta
11      endlet
12  process UI is
13      let UI : ui, E, V : str,
14      in
15          execute(gui, UI?) .
16          (   rec-event(UI, expr(E?)) .
17              snd-msg(compute, expr(E)) .
18              rec-msg(result, expr(E, V?)) .
19              snd-ack-event(UI, expr(E, V))
20          ) *
21          rec-event(UI, quit) .
22          snd-ack-event(UI, quit) .
23          shutdown("Goodbye!")
24      endlet
25  tool calc is { command="calc" }
26  tool gui is
27      { command="wish-adapter -script ui.tcl" }
```

Figure 1. Excerpt of a TOOLBUS script

note command from all processes that had previously issued a subscribe command on the corresponding label.

Processes use handshaking to communicate with tools. The tool-to-process communication can be either a send-event message (notifies an event) or a send-value message (sends a value), while the communication from a process to a tool can be one of the following three commands: snd-eval (evaluation request), snd-do (action request, i.e. without return value), or snd-ack-event (acknowledges a previous event).

All the communication commands may have typed parameters and return results. To distinguish between input and output parameters, the former are decorated with the symbol ? as suffix.

Let us illustrate all these concepts with a simple example.

Example 1 Figure 1 presents the definition of a tool bus gluing a graphical user interface and the (command-line) UNIX calculator calc. The tool bus is composed of these two tools and the related (two) processes. The first process, named CALC (cf. lines 01–11) mediates requests for numeric computations to a command-line calculator: it spawns the calculator tool and assigns the corresponding identifier to Tid (line 04), then repeatedly receives a message on channel compute and assigns its value to E (line 05), sends it for evaluation to the tool (line 06), gets the answer in variable V (line 07), forwards it along channel result (line 08), and also broadcasts it to any interested party (line 09). The second process, called UI (cf. lines 12–24), is responsible for getting expressions to be calculated from the user: it spawns a graphical user-interface gui tool (line 15), and then repeatedly receives expressions from the interface, transmits them to the calculator, gets the corresponding value, and forwards it back to the user interface (lines 16 to 20), until it gets a command to quit the application (line 21). Finally, the last three lines associate the toolbus actors with actual programs.

2.2. THE LAZY APPROACH TO SMT-SOLVING

Before getting to a detailed explanation of the classic SMT-solving algorithm, we recall the main concepts and properties of first-order logic employed in this approach.

2.2.1. First-order logic: We assume the usual syntactic (such as signature, variable, constant and function symbol, term, atom, literal, formula, sentence, and substitution) and semantic notions (such as interpretation, model, satisfiability, validity, logical consequence, and theory) of first-order logic (see, e.g., [19]). The symbol $=$ is a predefined logical constant. If l and r are terms, then the atom $l = r$ is an *equality* and the literal $l \neq r$ is a *disequality*.

In this paper, we consider *first-order theories* (i.e. set of first-order sentences) *with equality*, meaning that the predefined logical constant $=$ is always interpreted as a reflexive, symmetric, transitive relation which is also a congruence. Let $\mathcal{T}$ be a theory. A formula φ is *satisfiable in* $\mathcal{T}$ if it is satisfiable in a model of $\mathcal{T}$. The *satisfiability problem* for $\mathcal{T}$ amounts to checking whether any given finite and quantifier-free conjunction (or, equivalently, finite set) of literals is satisfiable in $\mathcal{T}$. A *satisfiability procedure* for $\mathcal{T}$ is an algorithm capable of solving the satisfiability problem of $\mathcal{T}$. The satisfiability of a quantifier-free formula φ can be reduced to the satisfiability of several conjunctions of literals by converting φ to disjunctive normal form (DNF), splitting on disjunctions, and then solving the resulting satisfiability problems. For this reason, by abuse of language, we talk about satisfiability problem when considering the problem of establishing the satisfiability of arbitrary quantifier-free formulas. Indeed, the conversion to DNF may result in an exponential blow-up of the size of the formula. A much more efficient way, in practice, to tackle this problem is described in the rest of this section. The problem of checking the satisfiability of arbitrary quantifier-free formulas in $\mathcal{T}$ is NP-hard, as it subsumes the problem of checking the satisfiability of

function *check_unsat* ($\mathcal{T}$: *theory*; φ: *ground formula*)
  $\varphi^b \longleftarrow fol2prop(\varphi)$
  while $sat_{\mathbb{B}}(\varphi^b)$ **do begin**
    $\beta^b \longleftarrow pick\_assignment(\varphi^b)$
    $(\rho, \pi) \longleftarrow sat_{\mathcal{T}}(prop2fol(\beta^b))$
    if $\rho =$ sat **then return** sat
    $\varphi^b \longleftarrow \varphi^b \wedge \neg fol2prop(\pi)$
  end
  return unsat

Figure 2. The core algorithm of a lazy SMT solver

Boolean formulas.

2.2.2. SMT-solving: An SMT solver takes as input a quantifier-free formula φ and it is capable of checking its (un-)satisfiability in a certain theory $\mathcal{T}$. The lazy approach to build SMT solvers consists of using a Boolean enumerator and a decision procedure for $\mathcal{T}$. The former enumerates all satisfying Boolean assignments of φ (its atoms are considered as Boolean variables) and then the decision procedure checks if each assignment is satisfiable (or not) in $\mathcal{T}$. Indeed, several refinements are needed to make this integration to work efficiently in practice as we may end up invoking the decision procedure exponentially many times in the number of atoms of φ. In the following, we give an abstract view of a lazy SMT solver based on the enumeration of (total) Boolean assignments and the use of "theory conflict clauses" to prune the search space of the Boolean solver. Although this will be sufficient to understand the rest of the paper, we point the reader to, e.g., [6] for more realistic versions of the lazy architecture.

We assume the availability of two simple functions. The first is the *propositional abstraction* $fol2prop$ function, i.e. a bijective mapping from atoms to Boolean variables, which is homomorphically extended to arbitrary Boolean combination of atoms. The second is the *refinement* $prop2fol$ function, which is the inverse of $fol2prop$. In the following, sat and unsat denote the possible values returned by a satisfiability procedure; β^b is used to denote a Boolean assignment; π is used to denote a conjunction (or, equivalently, a set) of literals and π^b its Boolean abstraction; in general, we use the superscript b to denote Boolean expressions.

Figure 2 presents a simple version of the core algorithm underlying any lazy SMT solver. The algorithm enumerates the (total) truth assignments for the Boolean abstraction of φ and checks for satisfiability in $\mathcal{T}$. It concludes satisfiability if an assignment is satisfiable in $\mathcal{T}$ or returns unsatisfiable, otherwise. The function call $sat_{\mathbb{B}}(\varphi^b)$ establishes whether the Boolean formula φ^b is satisfiable or not. The function $pick\_assignment$ returns a total assignment to all atoms in φ or equivalently, to all Boolean variables in $\varphi^b = fol2prop(\varphi)$. The function call $sat_{\mathcal{T}}(\beta)$ detects if the set β of literals is satisfiable in the background theory $\mathcal{T}$; if so, it returns $(\mathsf{sat}, \emptyset)$; otherwise, it returns (unsat, π), where $fol2prop(\pi) \subseteq \beta^b$ and π is an unsatisfiable set in $\mathcal{T}$, called a *theory conflict set*. The negation of $fol2prop(\pi)$ is a *theory conflict clause* and it is used to eliminate—at once—all (total) Boolean assignments sharing the same Boolean abstraction of the theory conflict set π. Indeed, when $fol2prop(\pi) = \beta^b$, we end up enumerating all possible Boolean assignments of φ^b and performances are likely to be poor. So, in practice, computing conflict sets is the key to speed-up the performances of $check\_unsat$.

3. DISTRIBUTED SMT SOLVING

In order to design a distributed version of the algorithm in Figure 2, a simple—yet promising—idea would be to modify $check\_unsat$ so as to consider $n \geq 2$ Boolean assignments at the same time and let n instances of the decision procedure (encapsulated in the function $unsat_{\mathcal{T}}$) be executed concurrently on the n Boolean assignments. In this way, we can hope to significantly reduce the overall running time of the reasoning system. Indeed, to make this observation practical, particular care must be put in choosing a suitable value for n. In the remaining of this section, we develop this idea by using the TOOLBUS architecture.

3.1. OVERVIEW OF THE ARCHITECTURE

The first step in designing the distributed version of $check\_unsat$ is to identify the various tools (i.e. the modules in TOOLBUS terminology) that can be distributed over a network. In doing this, good software engineering suggests to take into account the following desiderata. First, the distributed version shall be scalable so as to take advantage of having a large or a small amount of available computing resources. Second, the communication overhead in the distributed version shall be minimal; this requires sharing of data between the different tools to be minimal. Third, the distributed and the sequential versions of the algorithm shall have as much code in common as possible, in order to facilitate code maintenance when new features are implemented or changes are made to the code.

We have therefore split haRVey into the following components, as illustrated in the interaction diagram depicted in Figure 3.

1. The module 'Interface' is responsible for receiving proof obligations from interested clients (not displayed in the figure) and returns the result of solving.

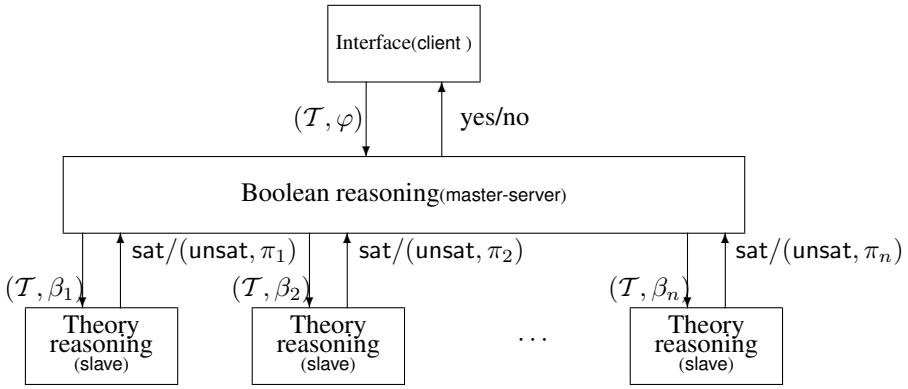


Figure 3. Architecture of the distributed version of $check\_unsat$ (cf. Figure 2)

2. The module 'Boolean reasoning' creates the Boolean abstraction of the given (quantifier-free) formula φ, it generates the Boolean assignments, and then refines and dispatch the resulting set of first-order literals to the available instances of the theory reasoning component.

   As the data structures necessary to handle propositional reasoning are quite complex and intertwined (be they BDDs or a SAT-solver), we chose to have a single instance of this component.

3. The module 'Theory reasoning' checks if a set of literals of the background theory $\mathcal{T}$ (corresponding to a Boolean assignment) is satisfiable or not.

   As it is possible to carry out several independent satisfiability checks, this component is the obvious candidate to be replicated in a distributed algorithm.

command =

The interaction between these components is modeled after two well-known architectural patterns in distributed programming: client/server and master/slave. More precisely, the interaction taking place between the modules 'Interface' and 'Boolean reasoning' follows the client/server pattern, while that occurring among the module 'Boolean reasoning' and the various instances of the module 'Theory reasoning' follows the master/slave pattern. Notice that the module 'Boolean reasoning' plays two roles at the same time: it is the server when interacting with the client 'Interface' and it is the master for each instance of the slave 'Theory reasoning.'

3.2. Description of the tool bus protocol

This section describes in detail the processes that form the interaction protocol between the different modules of the distributed version of the SMT-solver. The description starts with the main process and goes on with the different component sub-processes.

```
01  process Main is
02      let M : master-server,
03      in
04          execute(master-server, M?) .
05          (ConnectSlave(M)
06           +
07           Check(M)
08          ) *
09          rec-event(M, quit) .
10          shutdown("Checker is closed")
11      endlet
12  toolbus(Main)
```

Figure 4. The main process

3.2.1. The main process: The top-level process (Main) is depicted in Figure 4 and it corresponds to the box labelled with 'Boolean reasoning' in Figure 3. Main spawns an instance M of 'Boolean reasoning' (line 04)—which is a master-server in the sense it plays both roles (see above)—and then repeatedly (lines 05–08) shows one of the following two behaviours:

- the process ConnectSlave (line 05 and Figure 5) is activated when a connection request is received from an instance S (slave) of 'Theory reasoning' and
- the process Check (line 07 and Figure 7) is initiated on reception of a new quantifier-free formula from (the client) 'Interface.'

Finally, when M emits a quit event (i.e. when it has considered all assignments for satisfiability), the process Main successfully terminates (lines 09 and 10).

3.2.2. Establishing new master-slave connections: The process ConnectSlave (cf. Figure 5) is a sub-process of Main and it is responsible for the connection between

```
01  process ConnectSlave(M: master-server) is
02      let Pid : int, Name : str, S : slave
03      in
04          rec-connect(S?) .
05          create(Slave(S), Pid?) .
06          snd-do(M, slaveCreate(Pid))
07  endlet
```

Figure 5. Establishing a connection with a new slave

a new instance S of a slave (i.e. an instance of the module 'Theory reasoning') and the instance M of the tool master-server.

It sequentially receives a connection request from S (line 04), then creates an instance of the process Slave (described below) which is attached to S, gets the corresponding process identifier Pid (line 05), and it asynchronously notifies the master M that a new slave is available, sending a message parametrized with the value Pid of the process identifier (line 06). From the architectural viewpoint of Figure 3, the execution of this process corresponds to the creation of a new box labelled with 'Theory reasoning' (at the bottom of the Figure) and the establishment of a communication channel between this box and the box marked 'Boolean reasoning.'

3.2.3. Interface with instances of the first-order reasoning tool: The process Slave (cf. Figure 6) is the wrapper around the decision procedure for the background theory $\mathcal{T}$, which is required for this to be used in the TOOLBUS architecture. Slave takes as input the identifier S for an instance of 'Theory reasoning' and it is responsible for handling two types of events.

1. Requests for satisfiability checking in the background theory $\mathcal{T}$ are received from Check via a folCheckUnsat message (line 08) and are then dispatched to S (lines 09 and 10). The result (i.e. either sat or unsat together with a conflict set, if the case) is then sent back through a folCheckUnsatResults message (line 11).

2. Initialization requests, parametrized with the background theory $\mathcal{T}$, are forwarded to S, via a folInit message (lines 13 and 14). It is useful to parametrize initialization requests by $\mathcal{T}$ as an SMT solver may feature decision procedures for several background theories. However, notice that $\mathcal{T}$ is the same for a given quantifier-free formula. We will see in Section 3.4 that this parameter may play an important role when considering a background theory obtained as the union of several simpler theories.

3.2.4. Handling proof obligations: Process Check (cf. Figure 7) mediates the communications and describes the interactions between the three types of tools. Most importantly, it orchestrates the various activities required to check the satisfiability of a given quantifier-free formula. This can be decomposed in two phases. First, Check accepts a connection request from a client tool C (line 08), sends it the message propCheckUnsat, and gets the parameters of the satisfiability problem: the background theory and goal formula (lines 09 and 10), which are then forwarded to the master-server tool M (line 11). The process then enters in the second phase, which consists of the loop at lines 12–19, until the process gets notified by M that satisfiability checking has been completed (line 20). Then, it forwards the result (namely, sat or unsat) to the client and terminates it (lines 21–22). The body of the loop at lines 12–19 is a choice between two behaviours:

- lines 12—15: M can send out a new assignment to be checked for unsatisfiability by some available slave tool S. S is initialized and notified with the new satisfiability problem. Afterwards, M is sent an acknowledgement message, as soon as the activity of satisfiability solving has been started.

- lines 17–18: a slave tool S may return the answer to a satisfiability problem that has been previously sent out. The result is then forwarded to M.

The activity of satisfiability checking in the slave tools is done asynchronously with respect to the master tool. It is up to the master to create new assignments and dispatch them to slaves that have previously been connected to the tool bus. A detailed description of the internal structure of the master-slave tool is presented in the next section.

3.3. DESCRIPTION OF THE COMPONENTS

In this section, we describe the computations carried out in each of the three modules depicted in Figure 3. The module 'Interface' (client) performs simple operations on the input satisfiability problem, such as recognizing the background theory $\mathcal{T}$ and the quantifier-free formula ϕ specified in the input satisfiability problem (besides low level activities such as parsing); it is also responsible of performing the Boolean abstraction/refinement step by invoking the functions $fol2prop$ and $prop2fol$, respectively (see Section 2.2). The module 'Theory reasoning' (slave) simply invokes the function $unsat_{\mathcal{T}}$ (see again Section 2.2) upon reception of a message labelled folCheckUnsat.

For the 'Boolean reasoning' component (master-server), the situation is more complex. Its main activity is to generate several *distinct* Boolean assignments (satisfying the abstraction of the input formula) and dispatch each one of them to a slave (i.e. an instance of the mod-

```
01    process Slave(S: slave) is
02        let Assignment, Theory: term,
05            ProofStatus: int,
06            ProofLiterals: term
07    in
08    (   (rec-msg(folCheckUnsat(S, Assignment?)) .
09         snd-eval(S, folCheckUnsat(Assignment)) .
10         rec-value(S, folCheckUnsat(ProofStatus?, ProofLiterals?)) .
11         snd-msg(folCheckUnsatResult(S, ProofStatus, ProofLiterals)) )
12        +
13        (rec-msg(folInit(S, Theory?)) .
14         snd-do(S, folInit(Theory)) )
15    ) * delta
16    endlet
```

Figure 6. Process mediating communications with a slave tool

ule 'Theory reasoning') as soon as it becomes idle, in order to minimize latency. To avoid useless computations, it is crucial that the master-server fairly enumerate the Boolean assignments, i.e. without reconsidering the same two or more times. To meet all these requirements, a Boolean solver and some auxiliary data structures monitoring the progress in the satisfiability solving activity are used.

Efficient implementations of Boolean solvers are available, using either carefully engineered variants of the Davis-Putnam-Logeman-Loveland (DPLL) algorithm [12] (SAT-solvers) or Binary Decision Diagrams (BDD) packages. In this paper, we only consider *complete* solvers, i.e. based on algorithms that are always able to establish the satisfiability or the unsatisfiability of any Boolean formula SAT-solvers are based on the incremental construction of a single satisfying assignment while pruning the search space by using the inconsistencies derived by each truth value assigned, one after the other, to Boolean variables. BDDs compactly encode the disjunctive normal form of a formula, thereby representing all its possible satisfying Boolean assignments. So, SAT-solvers are apt to tackle very large satisfiability problems because they scale up much more significantly than BDDs, which suffer from exponential blow-up in space; but SAT-solvers are not designed to compute several Boolean assignments of a certain formula at once. On the contrary, it is straightforward to extend a BDD package with the capability of generating several distinct Boolean assignments as all possible assignments are readily available.

For simplicity, we have chosen BDDs as the core technique underlying the Boolean enumerator of multiple assignments required by master-server. Also, since we are more interested in SMT-solving for software verification, the issue of the dimension of the formulas to be refuted is less important as their Boolean structure is usually less important than for other (e.g., hardware) verification problems.

In order to avoid reconsidering several times the same Boolean assignment, it is sufficient to assume that the function $pick\_assignment$ (see Section 2.2) is fair, i.e. it does not return twice the same Boolean assignment, and that the master-server maintains some data structures to track the progress in the satisfiability solving activity: a set S containing the available slaves, a set $B \subseteq S$ containing those slaves which are busy, the background theory $\mathcal{T}$ of the current satisfiability problem, and a BDD of a formula φ^b representing the assignments that have not yet been checked for unsatisfiability.

The pseudo-code of the 'Boolean reasoning' module is shown in Figure 8. The set S of slaves is initialized and the event handling loop is started, namely $HandleEvents()$, which is supposed to handle the following messages. We omit the pseudo-code for $HandleEvents()$ as its implementation is automatically generated by the ToolBus.

- The message slaveCreate happens when a new instance s of the the 'Theory reasoning' component connects to the tool bus (line 06 of Figure 5). The routine handling this message simply adds the instance s to the set S of available slaves (see Figure 9).

- The message propCheckUnsat occurs when the 'Interface' component (client) has sent a new satisfiability problem to the tool bus (line 11 of Figure 7). The routine handling this message is given in Figure 10. First it initializes the satisfiability problem (by considering the background theory $\mathcal{T}$ and the quantifier-free formula φ) and then invokes the auxiliary routine $dispatch()$, whose pseudo-code is

```
01 process Check (M:master-server) is
02     let C: client, S: slave,
03         CTheory, CGoal: term,
04         MAssignment, MTheory: term,
05         SProofStatus: int, SProofLiterals: term,
06         FinalResult: int,
07     in
08         rec-connect (C?) .
09         snd-eval(C, propCheckUnsat) .
10         rec-value(C, propCheckUnsat(CTheory?,CGoal?)) .
11         snd-do(M, propCheckUnsat(CTheory,CGoal)) .
12         (    (rec-event(M, folCheckUnsat(S?,MAssignment?, MTheory?)) .
13               snd-msg(folInit(S,MTheory)) .
14               snd-msg(folCheckUnsat(S,MAssignment)) .
15               snd-ack-event(M,folCheckUnsat(S,MAssignment)) )
16         +
17              (rec-msg(folCheckUnsatResult(S?,SProofStatus?,SProofLiterals?)) .
18               snd-do(M,folCheckUnsatResult(S,SProofStatus,SProofLiterals)) )
19         ) *
20         rec-event(M, checkEnd(FinalResult?)) .
21         snd-do(C, checkEnd(FinalResult)) .
22         snd-terminate(C,FinalResult)
23     endlet
```

Figure 7. Process handling verification requests

function *master_server_main* ()
  B, S $\longleftarrow \emptyset, \emptyset$
  $HandleEvents()$

Figure 8. The main routine of the master-server

depicted in Figure 11, which is responsible for dispatching new assignments to idle slaves. This is implemented by the loop choosing an idle slave (if any) and considering a not yet dispatched assignment (by re-using the function $pick\_assignment$ of the sequential version of the SMT-solving algorithm, cf. Fig. 2), if any Boolean assignments are left to be considered (by invoking the function $has\_assignment$). Finally, after updating the set of busy slaves, the new satisfiability problem is dispatched to the selected (idle) slave. Otherwise, no assignment has been shown unsatisfiable (see message folCheckUnsatResult below) and the result is that the input formula is unsatisfiable in $\mathcal{T}$.

- The message folCheckUnsatResult happens when a slave s returns the result of invoking the function $unsat_{\mathcal{T}}$ on a dispatched assignment (line 11 of Figure 6), i.e. either sat or unsat. The routine handling

function *slaveCreate* (s: slave)
  $S \longleftarrow S \cup \{s\}$

Figure 9. The routine handling slaveCreate messages.

function *propCheckUnsat* (O: *options*, $\mathcal{T}$: *theory*, g: *formula*)
  $\varphi \longleftarrow g$
  $\mathcal{T} \longleftarrow T$
  dispatch()

Figure 10. The routine handling propCheckUnsat messages.

this message is given in Figure 12. First, the slave s becomes idle (i.e. it is deleted from the set B of busy slaves). Then, the outcome of a satisfiability problem is tested: if the assignment is unsatisfiable, then the (Boolean abstraction of the) conflict set π is used to prune φ^b (similar to what was done in the sequential case in Fig. 2) and a new Boolean assignment is considered (cf. the invocation to the function $dispatch()$, see also Fig. 11 above). Otherwise, the whole satisfiability solving activity is terminated with a checkEnd message. In this second case, the conclusion is that the formula φ is satisfiable in $\mathcal{T}$.

```
function dispatch ()
    if sat_B(φ^b) = sat then
        while (B \ S ≠ ∅) ∧ has_assignment(φ^b) do
            let s ∈ (B \ S) and
                β = prop2fol(pick_assignment(φ^b))
            in
                B ⟵ B ∪ {s}
                send-event(folCheckUnsat(s, β, T))
            end
        end
    else
        send-event(checkEnd(unsat))
    end
```

Figure 11. Handling propCheckUnsat messages: the auxiliary routine *dispatch*

```
function folCheckUnsatResult (s: slave,
                              ρ: {sat, unsat},
                              π: formula)
    B ⟵ B − {s}
    if (ρ = unsat) then
        φ^b ⟵ φ^b ∧ ¬π^b
        dispatch()
    else
        send-event(checkEnd(unsat))
    end
```

Figure 12. The routine handling folCheckUnsatResult messages

3.4. Flexibility of the approach: Delayed Theory Combination

Program verification often requires to verify proof obligations that are expressed as first-order formulas over a combination of diverse theories that reflect the different data and specification constructs used in program design and implementation artifacts: fragments of arithmetics, set theory, array theory, etc. From a practical viewpoint, such formulas are in expressed in a theory resulting from the union of the component thoeries.

The proposed architecture for the distributed version of an SMT solver can easily be adapted to handle satisfiability problems in combinations of theories, i.e. when the background theory $\mathcal{T}$ is obtained as the union of several simpler theories. In the following, we assume that $\mathcal{T}$ is the union of two theories $\mathcal{T}_1$ and $\mathcal{T}_2$; the generalization to more than two theories is straightforward. In order to precisely talk about combination of satisfiability procedures, we need to introduce some basic notions about unions of theories (for more details, the reader is pointed to, e.g., [25]).

3.4.1. Combination: Let $\mathcal{V}$ be a (finite) set of variables. An *identification* over $\mathcal{V}$ is an idempotent substitution from $\mathcal{V}$ to $\mathcal{V}$. Any identification σ over $\mathcal{V}$ defines a partition of $\mathcal{V}$ and identifies all the variables in the same equivalence class of the partition with a representative of that class. If σ is an identification over $\mathcal{V}$, then $\widehat{\sigma_=}$ ($\widehat{\sigma_\neq}$) is the conjunction of equalities (disequalities, resp.) of the form[1] $x\sigma = y\sigma$ ($x\sigma \neq y\sigma$, resp.) for $x, y \in \mathcal{V}$. Intuitively, $\widehat{\sigma_=}$ expresses the fact that any two variables identified by an identification σ must take identical value, while $\widehat{\sigma_\neq}$ expresses the fact that any two variables not identified by σ must take distinct values. Hence, the formula $\widehat{\sigma_=} \wedge \widehat{\sigma_\neq}$ (denoted with $\widehat{\sigma}$) faithfully represents the identification σ over $\mathcal{V}$.

A term in the union of $\mathcal{T}_1$ and $\mathcal{T}_2$ is an *i-term* if it is a variable or it has the form $f(t_1, ..., t_n)$, where f is in the signature of $\mathcal{T}_i$, for $i = 1, 2$. According to this definition, any variable is both a 1-term and 2-term. A non-variable sub-term s of an i-term t is *alien* if s is a j-term, where $i, j \in \{1, 2\}$ and $i \neq j$. An atom (literal) is *i-pure* if it contains only i-pure terms and its predicate symbol is either in the signature of $\mathcal{T}_i$ or is $=$, for $i = 1, 2$.

In order to re-use the satisfiability procedures for $\mathcal{T}_1$ and $\mathcal{T}_2$ (which can only handle sets of literals in the respective signatures) to solve the satisfiability problem in $\mathcal{T}$, i.e. in the union of $\mathcal{T}_1$ and $\mathcal{T}_2$, we need to perform a suitable pre-processing step, called *purification*. We purify a conjunction φ of literals in the union of $\mathcal{T}_1$ and $\mathcal{T}_2$ into a conjunction $\varphi_1 \wedge \varphi_2$, where φ_i is a conjunction of i-pure literals, for $i = 1, 2$. This is done by replacing each alien sub-term t with a "fresh" (i.e. not occurring in φ) variable x and adding the equality $x = t$ to the resulting formula. Indeed, purification terminates as there are only finitely many subterms in φ and it produces an equisatisfiable formula. The variables shared by φ_1 and φ_2 are called *interface variables* in $\varphi_1 \wedge \varphi_2$.

The non-deterministic version of the Nelson-Oppen combination schema [24] can be summarized as follows:

1. purify φ into the conjunction $\varphi_1 \wedge \varphi_2$, where φ_i contains only i-pure literals, for $i = 1, 2$;
2. guess an identification σ over the interface variables in $\varphi_1 \wedge \varphi_2$;
3. if $\varphi_i \wedge \widehat{\sigma}$ is satisfiable in $\mathcal{T}_i$ for both $i = 1$ and $i = 2$, then φ is satisfiable in the union of $\mathcal{T}_1$ and $\mathcal{T}_2$. Otherwise, go back to step 2 and consider another identification of variables (if any);
4. if no more identification of interface variables must be considered and no satisfiability has been detected at step 2, then φ is unsatisfiable in the union of $\mathcal{T}_1$ and $\mathcal{T}_2$.

[1] We write the application of a substitution in post-fix form.

The termination of the schema is obvious as there are only finitely many identifications to be considered over a finite set of interface variables. Its soundness can be shown under the assumption that $\mathcal{T}_1$ and $\mathcal{T}_2$ are signature disjoint, i.e. their signatures have no symbols in common except for the predefined logical constant $=$, and stably-infinite, i.e. for every satisfiable formula φ_i, there exists a model of $\mathcal{T}_i$ whose domain is infinite and which satisfies φ_i, for $i = 1, 2$ (see, e.g., [25]). Examples of stably-infinite theories are the theory of uninterpreted function symbols, the theory of Linear Arithmetic (both over integers and rationals), and the theory of arrays.

The non-deterministic schema above can be turned into a deterministic procedure by case-splitting on the formulas faithfully representing all the identifications over the shared variables and the satisfiability procedures for the component theories can be used to check the satisfiability of each pure conjunction of the split. Indeed, the resulting procedure is far from efficient. A much more efficient way, *in practice*, to tackle this problem has been proposed in [6] and it is briefly described in the rest of this section.

3.4.2. Delayed Theory Combination: The Delayed Theory Combination (DTC) schema does not require the direct combination of the procedures for $\mathcal{T}_1$ and $\mathcal{T}_2$ as in the Nelson-Oppen schema. Instead, DTC couples the available Boolean solver with each available procedure separately. The Boolean solver enumerates the Boolean assignments of the input formula φ extended with a conjunction $\widehat{\sigma}$ of equalities and disequalities, faithfully representing one of the possible identifications over the interface variables. This is obtained by adding to the set of Boolean variables abstracting the atoms in φ, the set of all equalities that can be formed over the shared variables. (It is possible to generalize the computation of the set of shared variables from conjunctions of atoms to arbitrary quantifier-free formulas, see [6] for details.) As a consequence, any Boolean assignment is of the form $\beta_1^b \wedge \beta_2^b \wedge \widehat{\sigma}^b$, where β_i is a conjunction of atoms in the theory $\mathcal{T}_i$, for $i = 1, 2$, and $\widehat{\sigma}$ is a conjunction of equalities and disequalities over the interface variables. Then, DTC invokes each available satisfiability procedure for $\mathcal{T}_i$ on $\beta_i \wedge \widehat{\sigma}$ ($i = 1, 2$): if both return satisfiable, then the satisfiability of φ in the union of $\mathcal{T}_1$ and $\mathcal{T}_2$ is derived; otherwise, at least one of the procedures detected an unsatisfiability, the (Boolean abstraction of the) negation of the corresponding conflict set is added to φ^b, and a new Boolean assignment of the resulting Boolean formula is considered, if any. When the Boolean formula becomes unsatisfiable, φ is declared unsatisfiable in the union of $\mathcal{T}_1$ and $\mathcal{T}_2$. DTC can be implemented by a simple modification of the core algorithm underlying an SMT solver depicted in Figure 2. The argument underlying its correctness is an adaptation of the correctness of the algorithm in Figure 2 and the completeness of the non-deterministic Nelson-Oppen schema (see [6] for details).

3.4.3. Distributed Delayed Theory Combination: As it is possible to derive an implementation of DTC from the algorithm for SMT solving in one theory, it is straightforward to derive a distributed version of DTC by modifying the distributed SMT solver described above (cf. Figure 3). The only delicate point is the handling of the messages returned from the 'Theory reasoning' modules, as they run asynchronously and there is no guarantee that their results are received in the right order by the 'Boolean reasoning' module. More precisely, we must make sure that the answers to the satisfiability problems $\beta_1 \wedge \widehat{\sigma}$ and $\beta_2 \wedge \widehat{\sigma}$ deriving from the same assignment $\beta_1 \wedge \beta_2 \wedge \widehat{\sigma}$ are considered together. This is done by decorating each $\beta_i \wedge \widehat{\sigma}$ with the same time-stamp t and propagating t to the result returned by the satisfiability procedure for $\mathcal{T}_i$ ($i = 1, 2$). Some more details follow.

The 'Boolean reasoning' module is extended so as to compute a purified version of the input formula φ, determine the set $\mathcal{V}$ of interface variables, and form all possible equalities over $\mathcal{V}$. Then, the Boolean solver is instrumented so as to generate Boolean assignments over the atoms in the input formula and the equalities over $\mathcal{V}$. When an assignment $\beta_1 \wedge \beta_2 \wedge \widehat{\sigma}$ is considered, it is split into two conjunctions of i-pure literals (namely, $\beta_1 \wedge \widehat{\sigma}$ and $\beta_2 \wedge \widehat{\sigma}$), each dispatched to an available 'Theory reasoning' module together with the identifier of the related theory $\mathcal{T}_i$, for $i = 1, 2$, and a common time-stamp t (i.e. the 'Boolean reasoning' module generates messages containing triples of the form $(\mathcal{T}_i, \beta_i \wedge \widehat{\sigma}, t)$) for $i = 1, 2$).

Each 'Theory reasoning' module is associated to one of the background theories $\mathcal{T}_i$ and it can accept a satisfiability problem only if it is in $\mathcal{T}_i$ ($i = 1, 2$). If the 'Theory reasoning' module refuses to handle a satisfiability problem, after being notified, the 'Boolean reasoning' module looks for another idle instance. Otherwise, the 'Theory reasoning' module sends the answer to the satisfiability problem back to the 'Boolean reasoning' module with the conflict set $\pi_{\mathcal{T}_i}$ (if the case), for $i = 1, 2$, and the received time-stamp t (i.e. the 'Boolean reasoning' module receives messages of one of the following forms: (unsat, t) or $(\mathsf{sat}, \pi_{\mathcal{T}_i}, t)$, where $\pi_{\mathcal{T}_i}$ is a conflict set in the theory $\mathcal{T}_i$ for $i = 1, 2$).

The 'Boolean reasoning' module maintains a set TS of time-stamps associated to sat results received from 'Theory reasoning' modules. As soon as a sat message is received, the corresponding time-stamp t is tested for membership in TS: if t is in TS, then the satisfiability of the input formula φ is derived; otherwise, t is added to TS. If an unsat message is received, it is also tested for membership in TS: if t is in TS, then t is deleted from TS;

otherwise, t is discarded. After the reception of an unsat message, independently of the fact that t is in TS or not, the (Boolean abstraction of the) negation of the associated conflict set $\pi_{\mathcal{T}_i}$ (for $i = 1, 2$) is passed to the Boolean solver.

4. EXPERIMENTS

As suggested in Section 3, we have implemented the distributed SMT solver by re-using the available functionalities of a sequential SMT solver. Our choice was haRVey [15], which is based on the algorithm depicted in Figure 2 and it is being developed by the first two authors and Pascal Fontaine. In haRVey, one is free to choose the implementation of the functions $sat_{\mathbb{B}}$ and $pick\_assignment$ (cf. Figure 2) by either BDDs [7] (the CMU BDD package by Long) or a SAT-solver (MiniSAT [18]). A peculiarity of haRVey is that the implementation of the function $unsat_{\mathcal{T}}$ is done by an automated (first-order) theorem prover, called the E prover [28], which is combined with a decision procedure for a fragment of Linear Arithmetic via the Nelson-Oppen combination schema [24]. The availability of an automated prover for full first-order logic makes it possible to handle also formulas containing quantifiers, as shown in [11], and this is particularly useful for software verification. So far, haRVey has been successfully applied to the verification of pointer-based programs [27], B specifications [11], static checking of automatically generated code for aerospatial applications [16] as well as array programs [14].

The distributed version of haRVey has been developed along the lines of Section 3 by using TOOLBUS version 0.24, the available BDD package in haRVey for the 'Boolean reasoning' module, and the E prover as the 'Theory reasoning' module. It took us 20 man hours to come up with the first running prototype which consists of 105 lines of TOOLBUS and 5900 lines of C code, mostly re-used from the sequential version of haRVey, whose source is 5300 lines of C code.

Experiments have been carried out over a 10 Mbps Ethernet network of desktop workstations in a normal working environment, where all the tools had to compete for resources with other user processes. The computing nodes had the following configuration: 2.0GHz Pentium IV processor, with 512MB of RAM, operated under Linux.

As benchmark problems, we have selected 50 proof obligations requiring several interactions between the Boolean solver and the satisfiability procedure from previous experiences with the sequential version of haRVey (namely, proof obligations generated from the B methodology [11], the verification of pointer-manipulating programs and Burns protocol [27]). The goal of our experiments was to understand if it were possible to significantly reduce the overall running time by invoking several instances of the satisfiability procedure concurrently.

To understand the rationale underlying the experiments, the following observation is useful. From Figure 2, recall that function $pick\_assignment$ chooses an assignment β which is checked for (un-)satisfiability in the background theory. If β is unsatisfiable, a conflict set π is computed and used to prune the search space by adding the Boolean abstraction of $\neg\pi$ to the input formula. In this way, all assignments satisfying π are no longer considered.

When using a BDD, to choose an assignment, $pick\_assignment$ recursively traverses it from the root to a true leaf (this can be done in time linear in the number of atoms occurring in the BDD). We have considered four ways to perform this traversal, according to how the sub-tree to traverse is chosen:

- the *rightmost* sub-tree is chosen (this is the simplest heuristic to implement and it is the one implemented in the sequential version of haRVey),
- either the right or the left sub-tree is *randomly* selected,
- the *zigzag* heuristic chooses alternatively the right or the left sub-tree, and
- in the *alternate* heuristics, the traversal proceeds by alternatively following the rightmost or the leftmost sub-tree.

Indeed, if two similar assignments are checked for satisfiability, one after the other (and found unsatisfiable), it is reasonable to expect that their conflict sets be similar (if not identical), thereby pruning essentially the same part of the search space of the Boolean solver. In a distributed environment, it seems desirable to choose a heuristics for $pick\_assignment$ that allows us to consider "different" assignments concurrently in the hope that their conflict sets will prune different portions of the search space, thereby (furtherly) reducing the execution time.

Guided by this observation, we present below two experiments. The former (Section 4.1) considers the sequential version of haRVey and studies its behaviour according to the four heuristics above to implement $pick\_assignment$. The latter (Section 4.2) consists of repeating the previous experiment with the distributed version in order to investigate the advantages of having several instances of the satisfiability procedure running concurrently. Indeed, the first experiment is conducted so as to be able to compute the speed-up of the distributed version and it is not very interesting *per se*.

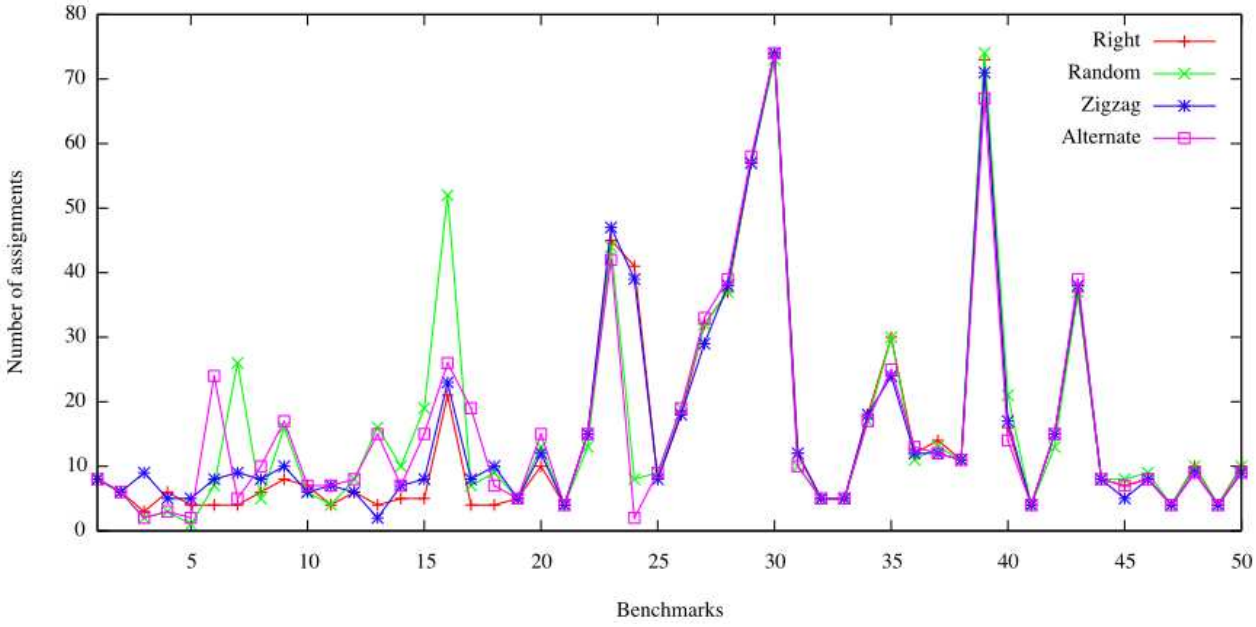


Figure 13. Sequential case: comparison of the assignment choice heuristics

4.1. SEQUENTIAL CASE

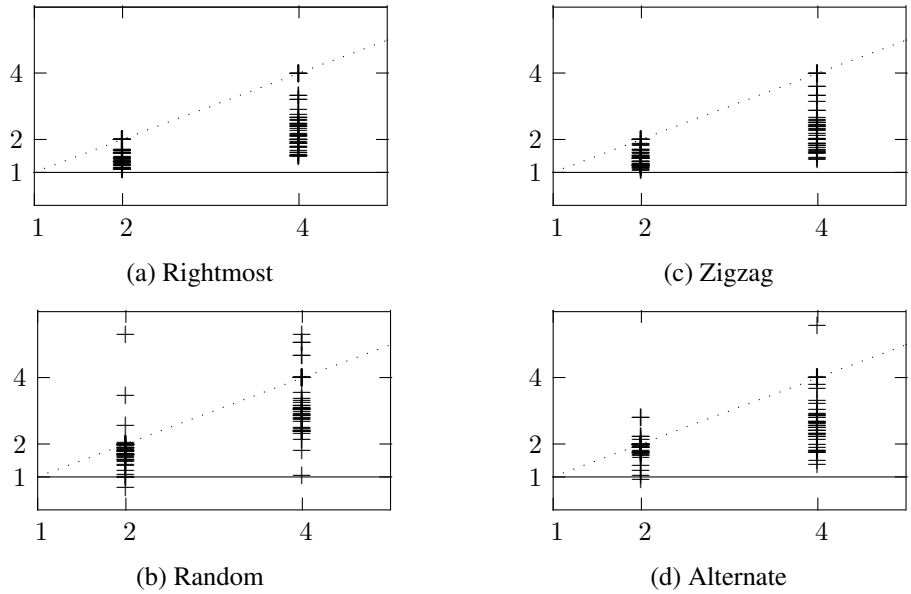
The graph in Figure 13 depicts the number of assignments (y-axis) that need to be considered for each of the four choice heuristics on the sequential version of haRVey (the x-axis shows the identifiers of the 50 proof obligations in our benchmark). 'Ri' stands for rightmost, 'Ra' for random, 'Zz' for zigzag, and 'Al' for alternate heuristic. As it is apparent, no approach dominates the others. We conclude that the assignment choice heuristics has no measurable impact on the performances of the sequential version of haRVey. Again, we emphasize that these results are not interesting *per se* but rather as a reference for the next experiment.

4.2. DISTRIBUTED CASE

We repeated the previous experiment with the distributed version of haRVey in two configurations: (up to) two or four slaves. For each of the 50 proof obligation in the benchmark, we measured the number of Boolean assignments that were considered with two and four slaves, using the four possible assignment choice heuristics (namely, rightmost, random, zigzag, and alternate). To measure the impact of distributing the workload, we computed the speed-up as follows:

$$S_n^h = \frac{n \cdot B_1^h}{B_n^h},$$

h is one of 'Ri', 'Ra', 'Zz', and 'Al', n is the number of instances of the 'Theory reasoning' modules (i.e. $n = 2$ or $n = 4$), B_1^h is the number of Boolean assignments considered in the sequential version of haRVey with h as assignment choice heuristic (cf. the values on the y-axis of the graph in Figure 13), and B_n^h is the number of Boolean assignments considered in the distributed version of haRVey with h as assignment choice heuristic. Notice that the value of B_n^h is averaged over several runs of the distributed version for each proof obligation. This is necessary because of the concurrent executions of the instances of the 'Theory reasoning' and the essentially random nature of low-level network protocols. In fact, when two instances of the 'Theory reasoning' module deliver their results concomitantly, 'Boolean reasoning' might receive them in a different order for two different executions on a given proof obligation. As a consequence, different sequences of conflict sets might be considered, thereby causing the 'Boolean reasoning' module to consider different assignments at the next iteration, which ultimately explains the difference in the number of generated assignments. In our experiments, we repeated each verification three times and report an average value for the number of Boolean assignment per proof obligation in the benchmark. The speed-ups are reported in Figure 14. For reference purposes, each diagram also contains two lines, corresponding to no speed-up (horizontal solid line, with $speedup = 1$), and linear speed-up (diagonal dotted line, with $speedup = n$). The closer the results are to the linear speed-up, the more efficient is the distributed version of haRVey. We can clearly visualize that, on our benchmark, the *random* assignment choice heuristic performs better than the three others (which present similar behaviors). A plausible explanation is that *rightmost*, *zigzag*



(a) Rightmost (c) Zigzag

(b) Random (d) Alternate

Figure 14. Distributed case: speed-ups for the four assignment choice heuristics

and *alternate* tend to generate similar successive assignments (in the case of *alternate*, every other assignment is generated on the same "region" of the graph representing the BDD), which have a higher probability to obtain the same (or similar) prunings of the search space, while this is not the case for *random*. Indeed, while in the sequential version, the $n+1$-th assignment is generated after the pruning of the search space with the conflict set generated from the n-th assignment, this is not necessarily the case in the distributed version. Therefore approaches that tend to generate different consecutive assignments will always tend to perform better in the distributed version.

Finally, note that, in some experiments, the distributed SMT solver achieves super-linear speed-ups. This happens when the conflict set associated to an assignment is so general that it prunes a (relatively) large part of the search space. This may happen for the class of formulas such that the value of a relatively small subset of the atoms causes unsatisfiability. Our experiment shows that the probability of achieving super-linear speed-ups is larger for the *random* assignment choice heuristic.

5. Conclusion

We have presented a distributed version of a lazy SMT solver, based on haRVey. This allows us to distribute the work-load over a network of workstations. The feature, unique to haRVey, that BDDs can be used to represent the Boolean structure of the formulas has greatly simplified the implementation of the distributed version.

The distributed algorithm has been prototyped by reorganizing the architecture of the sequential version of haRVey into three modules whose execution is concurrent: 'Interface,' 'Boolean reasoning,' and 'Theory reasoning.' The last module can be instantiated an arbitrary number of times in the distributed version. The implementation has been realized using the ToolBus architecture: a process algebra script describes the module interaction protocol and it is used to generate the code responsible for the communication and synchronization, as well as the interfaces that the modules implement to participate in the interaction. The extension of the architecture to handle a combination of background theories by using a distributed variant of the DTC schema show the flexibility of our approach.

Experiments on a set of representative proof obligations of software verification problems show the possibility to obtain super-linear speed-ups of the distributed SMT solver over its sequential version, thereby showing the viability and the benefits of the proposed architecture.

In the future, we plan to use the interaction protocol as a basis to a grid-based approach to SMT solving (see, e.g., [9] for a grid-based approach to Boolean solving). We also envision to extend or adapt the proposed protocol so that the 'Boolean reasoning' module can be implemented by a SAT-solver, based on some existing parallel implementations of DPLL algorithms (see, e.g., [22, 20]).

Acknowledgments

This work was partially funded by INRIA/CASSIS project and CNPq projects 490084/2005-2, 477960/2004-

9 and 307597/2006-7.

REFERENCES

[1] Thomas Ball, Byron Cook, Shuvendu K. Lahriri, , and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proceedings of the 16th International Conference on Computer-Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 457–461. Springer-Verlag, 2004.

[2] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Proceedings of the 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 2002.

[3] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc 16th Intl. Conf. Computer Aided Verification (CAV'2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, 2004.

[4] J. A. Bergstra and Paul Klint. The discrete time ToolBus — a software coordination architecture. *Science of Computer Programming*, 31(2–3):205–229, 1998.

[5] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

[6] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient Theory Combination via Boolean Search. *Information and Computation*, 204(10):1493–1525, 2006. Special Issue on Combining Logical Systems.

[7] Randy Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-38(8):677–691, 1986.

[8] Randy Bryant, S. German, and M.N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1), 2001.

[9] Wahid Chrabakh and Rich Wolski. GridSAT: A Chaff-based distributed sat solver for the Grid. In *Proc. of the 2003 ACM/IEEE Conference on Supercomputing*, page 37, 2003.

[10] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[11] Jean-François Couchot, David Déharbe, Alain Giorgetti, and Silvio Ranise. Scalable automated proving and debugging of set-based specifications. *Journal of the Brazilian Computer Society*, 9(2):137–151, 2004.

[12] M. Davis, G. Lovemand, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[13] H.A. de Jong and P. Klint. Toolbus: The next generation. In F.S. de Boer, M. Bonsangue, S. Graf, and W.P. de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *LNCS*, pages 220–241. Springer, 2003.

[14] David Déharbe, Abdessamad Imine, and Silvio Ranise. Abstraction–driven verification of array programs. In *Proc. of the 7th Int. Conf. on Artificial Intelligence and Symbolic Computation*, Lecture Notes in Artificial Intelligence, pages 271–275. Springer Verlag, 2004.

[15] David Déharbe and Silvio Ranise. Light-weight theorem proving for debugging and verifying units of code. In *Proc. of the Int. Conf. on Software Engineering and Formal Methods (SEFM03)*, pages 220–228. IEEE Computer Society Press, 2003.

[16] David Déharbe and Silvio Ranise. Satisfiability solving for software verification. In NASA, editor, *IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation*, number CP-2005-212788 in Tech. Rep., 2005.

[17] David Deharbe, Silvio Ranise, and Jorgiano Vidal. Distributing the workload in a lazy theorem prover. In *Proc. XIII Brazilian Symposium on Formal Methods (SBMF'2005)*, pages 17–31. Brazilian Computer Society, 2005.

[18] N. Eén and N. Sörensson. Minisat, 6th international conference on theory and applications of satisfiability testing (sat'03). page 2003.

[19] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York-London, 1972.

[20] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel Multithreaded Satisfiability Solver: Design and Implementation. *Electr. Notes Theor. Comput. Sci.*, 128(3):75–90, 2005.

[21] Cormac Flanagan, Rajeev Joshi, Xinming Ou, , and James B. Saxe. Theorem proving using lazy proof explanation. In *Proceedings of the 15th International Conference on Computer-Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 355–367. Springer-Verlag, 2003.

[22] Malay K. Ganai, Aarti Gupta, Zijiang Yang, and Pranav Ashar. Efficient distributed sat and sat-based distributed bounded model checking. In *Correct Hardware Design and Verification Methods (CHARME 2003)*, volume 2860 of *Lecture Notes in Computer Science*, pages 334–347, 2003.

[23] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[24] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[25] S. Ranise, C. Ringeissen, and D.-K. Tran. Nelson-Oppen, Shostak, and the Extended Canonizer: a Family Picture with a Newborn. In *First Int'l. Symp. on Theoretical Computer Science (ICTAC'04)*, volume 3405 of *Lecture Notes in Computer Science*, pages 372–386, China, 2004. Springer.

[26] S. Ranise and C. Tinelli. Satisfiability Modulo Theories. *IEEE Magazine on Intelligent Systems*, 21(6):71–81, November/December 2006.

[27] Silvio Ranise and David Déharbe. Applying lightweight theorem proving to debugging and verifying pointer programs. *Electronic Notes in Theoretical Computer Science*, 86, 2003. Proceedings of 4th Intl. Workshop on First-Order Theorem Proving (FTP'03).

[28] Stephan Schulz. E—a brainiac theorem prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[29] M.G.J. van den Brand and P. Klint. Aterms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, 49:55–64, 2007.