Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 184 (2007) 21–37
www.elsevier.com/locate/entcs

Distributing the Workload in a Lazy Theorem-Prover

David Deharbe[1,2]

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Natal, RN, Brazil

Silvio Ranise[3]

Laboratoire Lorrain de Recherche en Informatique et ses Applications
Institut National de Recherche en Informatique et en Automatique
Nancy, France

Jorgiano Vidal[4]

Núcleo de Desenvolvimento de Software
Centro Federal de Educação Tecnológica do Rio Grande do Norte
Natal, RN, Brazil

Abstract

Automated theorem proving consists in automatically (i.e. without any user interaction) discharging proof obligations which arise when applying rigorous methodologies for designing critical software systems. Recent developements in the so-called *lazy* approach in the integration of Boolean satisfiability with decision procedures for decidable theories of first-order logic have provided new means to efficiently prove or refute such proof obligations. In this paper, we present the first (known) attempt to design a distributed version of lazy theorem proving on a network of computers so that the available processing power can be used more effectively and avoid that automated reasoning be the bottleneck of the application of formal methods. Experiments clearly show the viability and the benefits of the proposed approach.

Keywords: Automated theorem proving, SMT prover, distributed computing.

[1] This work has been possible partially supported by the Brazilian Research Agency, CNPq, grants number 500473/2003-0, 477960/2004-9 and 490084/2005-2.

[2] Email: david@dimap.ufrn.br

[3] Email: Silvio.Ranise@loria.fr

[4] Email: jorgiano@cefetrn.br

doi:10.1016/j.entcs.2007.03.013

1 Introduction

Formal verification tools and techniques are challenged by increasingly complex software systems. In particular, checking that a property is met by a system is the bottleneck in the application of virtually any formal design approach. One recurring approach to this problem consists of building a conservative abstraction of the system and check if it is satisfied. If this is the case, the original model also satisfies the property; otherwise, we are not allowed to conclude and a refinement step is undertaken. The abstract model is changed to take into account more details and it is checked if such a model satisfies the property of interest. This abstract-check-refine cycle is repeated until we are allowed to conclude or the available computational resources are exhausted. State-of-the-art model checkers for software verification (see e.g. [8,2]) are typical examples of this approach to verification.

These different verification activities can be carried out, at least in part, using so-called lightweight theorem provers, i.e. provers providing a high-degree of automation for selected classes of formulae in decidable theories of first-order logic (FOL). Lightweight theorem provers have also been used successfully to check consistency of formal specification artifacts [9] and extended static checking of software code [14]. The construction of such provers has been made possible by recent advances in the integration of highly efficient Boolean solvers (e.g. SAT solvers) and decision procedures for theories in FOL. This integration can be either *eager* or *lazy*. In the eager integration, the formulas are translated to propositional logic and decided using a SAT-solver (see, e.g. [6]). Their main limitation is w.r.t. *flexibility* since the decidable theory must obey severe restrictions for the translation to propositional logic to be possible. Lazy integrations abstract atoms of formulas in FOL to propositional letters so that boolean satisfiability solvers (SAT solvers), or Binary Decision Diagrams (BDDs), can be used to extract a satisfiable propositional assignment of the abstract formula (BDDs actually compactly represent all the satisfiable assignments). Such an assignment is then translated back to a conjunction of literals in FOL which can then be checked valid (or not) by the available decision procedure for the first-order to which they belong. If checked valid, the process is started again by considering a new propositional assignment. Otherwise, the process halts by reporting that the first-order input formula is not valid. The lazy approach is at the basis of several recently developed tools such as Zapato [1], Verifun [15], CVC Lite [3] and our tool haRVey [12]. The main advantage of lazy over eager integrations is a much higher degree of flexibility while maintaining efficiency. For example, heterogenous theories obtained by unions of "simple" theories can be considered by using well-known combination methods (see [21] for an overview). More recently, it has been shown that a carefully engineered implementation of a lazy integration can outperform the eager approach [17]. In the case of haRVey, an even greater flexibility is achieved by integrating an automated deduction engine that can be configured with finitely axiomatizable theories.

The paper presents the first distributed approach to lightweight theorem proving based on the following simple observation. If we consider n (> 1) propositional assignments at the same time and we invoke n instances of the decision procedures

on their first-order counterparts (as sketched above), we can hope to significantly reduce the overall running time of the reasoning system. 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 open question is the choice of the n propositional assignments that provide best results. As experimentation is a mean to validate answers to these questions, we have implemented a distributed version of haRVey which exploits the above observation. Our implementation uses the ToolBus architecture [4] which allowed us rapid prototyping. This paper presents an experimental analysis with this prototype on a set of representative proof obligations showing the viability of the proposed approach. In particular, we study the impact on performances of different approaches in choosing n propositional assignments to be considered at the same time.

Plan of the paper.

Section 2 gives some background material on the ToolBus architecture. Section 3 explains the basic algorithm underlying haRVey. It also explains how a distributed algorithm has been derived from the original sequential version. Section 4 describes our prototype implementation of a distributed variant of haRVey. Section 5 reports on the experiments. Finally, Section 6 concludes and sketches our research directions.

2 The ToolBus

The construction of a heterogenous, distributed system is a challenging task, both from a design and implementation point of views. The ToolBus provides an elegant solution to implement robust distributed systems, using a process algebra as language to describe the protocol between the different components, and a uniform, term-like, communication data type called Aterms, which is also used in haRVey to represent logic formulas. In this section, we provide the reader with just the information necessary to understand how haRVey components may be distributed and interconnected with the help of the ToolBus. Further details can be found in e.g. [4].

Programmers write a ToolBus script describing the intended interaction protocol between the components, called *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 the programming languages JAVA and C (adapters are also available for Perl, Python, Tcl/Tk and UNIX scripts).

A ToolBus script defines an interaction protocol between different (user-supplied) tools by means of a composition of processes. We will adopt the term *tool bus* to denote an instance of such 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. Moreover,

the create operator dynamically creates process instances; finally, execute and snd-terminate respectively spawns and aborts the execution of a tool instance.

The execution of a tool can be dispatched from within the tool bus, with the command execute, or can be started externally and issue 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-note command from all processes that had previously issued a subscribe command on the corresponding label.

Processes use handshaking to communicate with the 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 communcation 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 latter are decorated with the symbol ? as suffix.

To illustrate these concepts, Figure 1 presents the definition of a tool bus providing the glue between a graphical user interface and the command-line UNIX calculator calc. The tool bus is composed of these two tools and two processes. The first process, named CALC (defined lines 1–11) 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 compute (line 08), and also broadcasts it to any interested party (line 09). The second process, called UI, is responsible to get 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, both tools are configured with actual applications (lines 25 and 26).

3 The lightweight theorem-prover haRVey

haRVey [12] checks if a first order formula ψ (possibly containing quantifiers) is a logical consequence of a theory $\mathcal{T}$ which can be finitely axiomatized. For simplicity, in this paper, we will assume that ψ is of the following form $\forall x_1, ..., x_n.\phi$, where $x_1, ..., x_n$ are the only variables occurring in ϕ which is required to be quantifier-

```
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 { command = "wish-adapter -script ui.tcl" }
```

Fig. 1. Excerpt of a ToolBus script

free[5]. haRVey works by refutation, i.e. it negates the formula ψ and tries to show that this negation is $\mathcal{T}$-unsatisfiable, i.e. that models of $\mathcal{T}$ do not satisfy $\neg\psi$. Under the assumptions above on ψ, $\neg\psi$ is $\exists x_1, ..., x_n.\neg\phi$ and hence we consider the problem of determining the $\mathcal{T}$-unsatisfiability of $\neg\phi$ where each x_i is considered as a (Skolem) constant.

haRVey is based on a combination of Boolean solving and superposition theorem proving. The core algorithm of haRVey is presented in Figure 2. Let *fol2prop* be a bijective mapping from ground atoms of ϕ to boolean propositions, *abs* its homomorphical extension to ground formulas, and *prop2fol* the inverse mapping from boolean propositions to ground atoms. ϕ is first univocally abstracted to a propositional formula ϕ^a and a Boolean solver finds satisfiable assignments to this abstraction. Each such assignment β^a is then translated back to a conjunction $\ell_1 \wedge \cdots \wedge \ell_n$ of ground first-order literals, which can be checked for $\mathcal{T}$-(un)satisfiability, using a superposition-based reasoning wrapped into the function *fol_check_unsat*. This is actually implemented by simply feeding a third-party superposition-based

[5] Readers interested in the technical details on the elimination of quantifiers are referred to [9].

prover with the axioms of $\mathcal{T}$ and the set $\{\{\ell_i\}\}_{i=1,\ldots n}$ of unit clauses. A refinement of this schema which dramatically improves performances (based on the capability of generating suitable lemmas to *prune* the search space of the Boolean solver) is actually implemented in haRVey. The function *fol_check_unsat* is extended so that also a small unsatisfiable sub-formula of the input conjunction of ground literals is returned. More precisely, we assume that $\mathit{fol\_check\_unsat}(\mathcal{T}, \beta) = (\top, \pi)$ iff π is a sub-formula of β and π is $\mathcal{T}$-unsatisfiable. π is called the *proof* of the assignment β. Using π to constrain the formula ϕ, several assignments can be discharged at once, resulting in often dramatically better results.

function *check_unsat* (T: *theory*; ϕ: *formula*)
  $\phi^a \longleftarrow \mathit{abs}(\phi)$
  while $\phi^a \neq \bot$ **do begin**
    $\beta^a \longleftarrow \mathit{pick\_assignment}(\phi^a)$
    $(\rho, \pi) \longleftarrow \mathit{fol\_check\_unsat}(\mathrm{T}, \mathit{prop2fol}(\beta^a))$
    if $\rho \neq \top$ **then return** *no*
    $\phi^a \longleftarrow \phi^a \wedge \neg \mathit{fol2prop}(\pi)$
  end
  return *yes*

Fig. 2. The core algorithm of haRVey.

A key feature of haRVey with respect to other lazy theorem provers is the fact that the decision procedure is based on superposition-based automated deduction, which makes it possible to apply it to verify the validity of a formula in any finitely presentable background theory. This feature also provides the possibility to handle nicely the verification of formulas with quantifiers [9]. In the current developement version of haRVey, boolean satisfiability can be realized with BDDs [5] or the SAT-solver zchaff [18], and first-order reasoning is a combination a la Nelson and Oppen [19] of the superposition-based E prover [22] and a decision procedure for a fragment of linear arithmetics. So far haRVey has been successfully applied to the verification of pointer-based programs [20], B specifications [9], static checking of automatically generated code for aerospatial applications [13] as well as array programs [11].

4 Distributed Algorithm

By considering Figure 2, it is not difficult to see how to obtain a distributed version of the algorithm. The key idea is to consider n (> 1) propositional assignments concurrently and invoke n instances of the decision procedures on their first-order counterparts (as sketched above); 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.

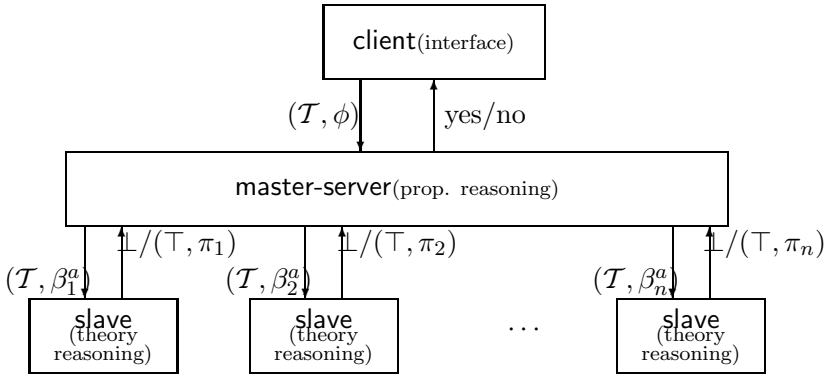

Fig. 3. Schematic architecture of distributed haRVey.

4.1 Overview of the architecture

The first step in designing the distributed version of haRVey is to identify the different components (or tools, in ToolBus terminology) that can be distributed over a network, with the following restrictions in mind:

(i) The distributed version shall be scalable so that it can take advantage of having a large or a small number of nodes.

(ii) The communication overhead in the distributed version shall be minimal, so the data shared between the different components shall be as little as possible.

(iii) The distributed and the sequential versions of the tool shall have as much code in common as possible, in order to limit maintenance problems.

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

interface: It is responsible for receiving proof obligations from interested clients and returns the result of the verification. Even though we could imagine having several instances of the interface, it only complicates the protocol and does not offer much insight on how a distributed lazy theorem prover performs.

propositional reasoning: The component will maintain the propositional abstraction of the given proof obligations, generate assignments, and send them to the instances of the first-order 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.

first-order reasoning: This is the component responsible to verify if assignments are indeed satisfiable or invalid. As it is necessary to carry out several independent verifications of this type, we identified that this component is candidate to be replicated in a distributed algorithm.

The interaction between these components is modeled after two well-known patterns of distributed programming. First, the interaction taking place between the interface and the propositional reasoning tool follows the *client/server pattern*, while the interaction occuring between the propositional reasoning component, and the

```
01    tool master-server is { command = "prop-reasoning" }
02    tool slave is { command = "fol-reasoning" }
03    tool client is { command = "reasoning" }
```

Fig. 4. Declaration of the components in the distributed haRVey tool bus.

first-order reasoning instances follows the *master/slave pattern*. Figure 4 presents the declaration of the three corresponding tools in the ToolBus script.

4.2 Description of the tool bus protocol

The main process.

The top-level process is presented in Figure 5. It spawns an instance M of the propositional reasoning tool master-server (line 04), and then repeatedly presents one of the two following behaviours:

- A connection request from an instance S of the first-order reasoning tool slave is handled in process ConnectSlave (line 05).
- The reception of a new proof obligation is dealt with in process Check (line 07).

This loop is exited when M emits a quit event (i.e. when it has completed the verification), at which point the process is stopped (lines 09 and 10).

Establishing new master-slave connections.

The process ConnectSlave (see Figure 6) is the sub-process of the main process responsible for the connection between a new instance S of the first-order reasoning tool slave and the instance M of the tool master-server. It sequentially receives a connection request from S (line 04), creates an instance of the process Slave (described below) which is attached to S, gets the corresponding process identifier Pid (line 05), and notifies asynchronously the master M that a new slave is available, sending a message parameterized with the value of Pid (line 06).

Interface with instances of the first-order reasoning tool.

The process Slave (see Figure 7) describes the part of the protocol related to interfacing one instance S of the slave tool responsible to carry out first-order reasoning. It is responsible for handling two types of events:

- Verification requests are emitted from the Check process via a folCheckUnsat message (line 08), are dispatched to be evaluated to the slave tool S (lines 09 and 10). The result is then sent back through a folCheckUnsatResults message (line 11).
- Initialization requests, parameterized with the set of theory axioms, are forwarded to S, via a folInit message (lines 13 and 14).

Handling proof obligations.

Process Check (see Figure 8) mediates the communications between the three types of tools to coordinate the verification of a proof obligation. In a first phase,

```
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)
```

Fig. 5. The main process

```
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
```

Fig. 6. Establishing a connection with a new slave

```
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
```

Fig. 7. Process mediating communications with a slave tool

it accepts a connection request from a client tool C (line 08), sends it the message propCheckUnsat and gets the parameters of the verification: theory axioms and goal formula (lines 09 and 10). These parameters are forwarded to the master-server tool M (line 11). The process enters then in the second phase, which is the main loop of the distributed algorithm (lines 12 to 19), until it gets signaled by M that

the verification has completed (line 20), then forwards the result to the client and terminates it (lines 21 and 22). The main loop is a choice between two behaviours:

- First (lines 12 to 15), M can emit a new assignment to be checked for unsatisfiability by some slave first-order reasoning tool S. S is then initialized and signaled this new verification task. M is sent an acknowledgement as soon as the verification has been started.
- Second, and conversely, (lines 17 and 18), a slave first-order reasoning tool S may return the result of the verification of a previously sent assignment. This result is then forwarded to M.

The verification 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 inner workings of the master-slave tool is presented in the next section.

```
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
```

Fig. 8. Process handling verification requests

4.3 Description of the components of distributed haRVey

This section provides details on the inner workings of each of the three tools that compose the distributed version of haRVey. The client interface and the first-order

reasoning slave basically consist in wrapping the corresponding functionalities implemented in the sequential version of haRVey. For instance, the slave has just to invoke the routine *fol_check_unsat* (as in Figure 2) when it receives a message labelled folCheckUnsat.

However, for the propositional reasoning master-server, the situation is not so simple. A new assignment has to be dispatched whenever a slave instance is idle. So the master-server must maintain the set S of the available first-order reasoning slaves, the set $B \subseteq S$ of the busy slaves, the theory $\mathcal{T}$ of the current proof obligation, a propositional formula ϕ to represent the assignments that have not yet been checked for unsatisfiability, and another propositional formula ψ to maintain the assignments that have not yet been dispatched to some slave.

The master-server must therefore be able to generate several different models from a propositional formula. Now, recall that haRVey can either use a SAT-solver or a BDD-based representation for the propositional reasoning. Contemporary SAT-solvers, based on evolutions of the Davis and Putnam algorithm [10], are not designed to provide several satisfying assignments and it would be necessary to make significant alterations to adapt an existing tool to our needs. In the case of BDDs, the situation is much simpler: each branch from the root node to the true leaf is a satisfying assignment (indeed a BDD is no more than a compact representation of all possible assignments). Moreover, although SAT-solvers are able to handle much larger formulas than BDDs, the complexity of the propositional structure in the proof obligations generated for software verification is usually simple enough to be handled easily with BDDs. Finally, using SAT-solvers require converting formulas to clausal normal form, introducing extra propositional variables, and eventually makes harder the reasoning in the background theories. Thus, in this first attempt at exploring the potential of distributed lazy theorem proving, we opted to use a BDD based representation.

The main routine of the master-server tool is shown in Figure 9. The set of slaves is initialized and the event handling loop is started. The following messages are handled in this loop (the implementation of the event-handling part of the code is automatically generated by the ToolBus):

slaveCreate The message happens when a new instance of the slave first-order reasoning tool connects to the tool bus (line 06 of Figure 6). The routine handling this message is given in Figure 10.

propCheckUnsat The message occurs when the client user-interface tool has sent a new proof obligation to the tool bus (line 11 of Figure 8). Figure 11 contains the routine handling this message: the state variables are updated with the proof obligation, and the auxilixary routine *dispatch* is invoked. This routine is responsible for dispatching new assignments to idle slaves and is detailed in Figure 12. It first checks that there are assignments that have not yet been checked. If this is the case, then for each idle slave, a new assignment is constructed (using the same routine as in the sequential algorithm) and dispatched. Otherwise, no assignments have been shown unsatisfiable (see next paragraph) and the result is that the given formula g is $\mathcal{T}$-unsatisfiable.

folCheckUnsatResult This message happens when a slave tool returns a result (line 11 of Figure 7). The corresponding routine is given in Figure 13. First, the outcome of the verification is tested: if the assignment is unsatisfiable, then the (smaller) unsatisfiable subset π of the assignment is used to prune both ϕ and ψ, as in the sequential case, and a new assignment is dispatched as explained above. Otherwise, the whole verification is ended with an checkEnd message. In this case, the result is that the formula g is $\mathcal{T}$-satisfiable.

5 Experimental results

The distributed version of haRVey, as described in the previous section, has been implemented and tested in a small network of workstations. We report on two experiments that we have carried out. First, we present in Section 5.1 different approaches to choose the assignment. As a matter of fact, as this choice has a direct impact on how much the propositional formula gets pruned, it may affect the efficiency of the verification. Second, in Section 5.2, we present the speed-up obtained with the distributed version of haRVey considering a varying number of available slaves.

These experiments have all been carried out in an 10Mbps Ethernet network, in a normal working environment, where all the tools had to compete for resources with other user processes. The examples composing the benchmarks have been collected from different protocol and software verification examples. Indeed, we used proof obligations generated from the B methodology [9], the formalization of the Burns mutual exclusion protocol, and the verification of pointer-manipulating programs [20]. This benchmark has been composed of proof obligations requiring several interactions between the propositional and first-order reasoning engines.

5.1 Assignment choice

We have developed several approaches to choose a (satisfying) assignment from the propositional abstraction of a formula. As distributed haRVey only uses BDDs, we have developed these approaches for BDDs. We recall that an assignment is represented in a BDD as a branch from the root node to the leaf node for the constant true. Traversing the BDD to find an assignment is straightforward and basically consists in recursing down the graph up to a node that has a true child node. The complexity is thus $O(n)$, where n is the number of atoms in the verified formula.

The goal of the assignment choice approaches is to find assignments such that, if they are shown unsatisfiable, their proof can be used to prune as large a number of assignments as possible. Indeed, if two similar assignments are verified, it is reasonable to expect that their proof would prune the same portion of the search space in the propositional representation. With these ideas in mind we developed four different approaches. The *rightmost* approach is the most simple, and was originally implemented in sequential haRVey: the BDD is recursively traversed down

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

Fig. 9. The main routine of the master-server.

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

Fig. 10. The routine handling slaveCreate messages.

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

Fig. 11. The routine handling propCheckUnsat messages.

function *dispatch* ()
  if ϕ **then**
    while $B - S \neq \emptyset \wedge \psi$ **do**
      let $s \in (B - S)$, $\alpha \leftarrow pick\_assignment(\psi)$ **in**
        $\psi \longleftarrow \psi \wedge \neg\alpha$
        $B \longleftarrow B \cup \{s\}$
        send(folCheckUnsat(s, α, $\mathcal{T}$))
      end
    end
  else
    send(checkEnd($\top$))
  end

Fig. 12. The auxiliary routine *dispatch*.

function *folCheckUnsatResult* (s: slave, r: *boolean*, π: *formula*)
  $B \longleftarrow B - \{s\}$
  if r **then**
    $\phi \longleftarrow \phi \wedge \neg\pi$
    $\psi \longleftarrow \psi \wedge \neg\pi$
    dispatch()
  else
    send(checkEnd($\bot$))
  end

Fig. 13. The routine handling folCheckUnsatResult messages.

by picking the right child. The *random* approach choses randomly one of the two child nodes. The *zigzag* approach choses alternatively the right or the left child. Finally, in the *alternate* approach, the traversal produces alternatively the rightmost or the leftmost branch.

Test	Ri	Ra	Zz	Al	Test	Ri	Ra	Zz	Al	Test	Ri	Ra	Zz	Al
01	8	8	8	8	02	6	6	6	6	03	3	2	9	2
04	6	3	5	3	05	4	1	5	2	06	4	7	8	24
07	4	26	9	5	08	6	5	8	10	09	8	16	10	17
10	7	6	6	7	11	4	4	7	7	12	6	8	6	8
13	4	16	2	15	14	5	10	7	7	15	5	19	8	15
16	21	52	23	26	17	4	7	8	19	18	4	9	10	7
19	5	5	5	5	20	10	13	12	15	21	4	4	4	4
22	15	13	15	15	23	45	44	47	42	24	41	8	39	2
25	8	9	8	9	26	18	19	18	19	27	32	32	29	33
28	37	37	38	39	29	57	57	57	58	30	74	73	74	74
31	12	10	12	10	32	5	5	5	5	33	5	5	5	5
34	18	17	18	17	35	30	30	24	25	36	12	11	12	13
37	14	13	12	12	38	11	11	11	11	39	73	74	71	67
40	16	21	17	14	41	4	4	4	4	42	15	13	15	15
43	37	37	38	39	44	8	8	8	8	45	7	8	5	7
46	8	9	8	8	47	4	4	4	4	48	10	10	9	9
49	4	4	4	4	50	10	10	9	9					

Table 1
Comparison of the assignment choice approaches (sequential case), where Ri stands for rightmost, Ra for random, Zz for zigzag, and Al for alternate.

Table 1 reports the number of assignments that need to be computed for each of these approaches on the sequential version of haRVey. No approach dominate the others, and we conclude that, in the sequential case, the assignment choice approach has no measurable impact on the average result. Indeed, when an assignment is found unsatisfiable, its proof is returned and is used to prune the search tree. This prevents other assignments with the same proof from being generated in the remainder of the search.

5.2 Performance in the distributed version

We repeated the previous experiment with the distributed version of haRVey, set up with two and four slaves. One important remark is that, due to the essentially random nature of low-level network protocols, the number of assignment checks to verify the original formula is *non-deterministic* in the distributed case. Indeed,

when two slaves deliver a result concomitantly, the master might receive them in a different order. This may result in different prunings, and cause the master to dispatch a different assignment at the next iteration, which ultimately explains the difference in the number of generated assignments. In our experiments, we repeated each verification several times and report here an average value of the measures.

Thus, for each of the 50 examples in our benchmark, we measured the number of dispatched branches with one (B_1), two (B_2) and four (B_4) slaves, using the four assignment choice approaches presented in the preceding Section. To illustrate the impact of distribution, we computed the estimated speedup $S_n = (n.B_1)/(B_n)$, where n is the number of slaves.

The measured speedups are reported in Figure 14. For reference purposes, each diagram also contains two lines, corresponding to no speedup (horizontal plain line, with $speedup = 1$), and linear speedup (diagonal dotted line, with $speedup = slaves$). The closer the results are to the linear speedup, the more efficient is the distributed algorithm. We can clearly visualize that, on our benchmark, the Random approach performs better than the three others, which present similar behavior. A plausible explanation is that Rightmost, Zigzag and Alternate tend to generate similar successive assignments (in the case of Alternate, every other assignment is generated on the same area of the BDD), which have a higher probability to realize the same or similar prunings, 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 tree with the proof generated from the n-th assignment, this is not necessarily the case in the distributed version; therefore approaches that tend to generate dissimilar consecutive assignments will always tend to perform better in the distributed version.

Finally, note that, in some experiments (mainly happening in those conducted with the Random approach), distributed theorem proving achieves super-linear speedups. This happens when the proof of unsatisfiability of an assignment is so general that it prunes a relatively large part of the search tree. 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 speedups is larger in the random choice assignment approach.

6 Conclusion

This paper presents how the lightweight theorem prover haRVey, which uses a lazy approach, can be distributed over a network of computers. The feature, unique to haRVey, that BDDs can be used to represent the propositional structure of the formulas to be verified has greatly simplified the realization of the distributed version.

The distributed algorithm has been prototyped by distributing the code of the sequential version of haRVey into three distinct tools: user interface, propositional reasoning, and first-order reasoning. The last tool can be instantiated an arbitrary number of times in the distributed algorithm. The implementation has been realized using the software architecture ToolBus: a process algebra script describes

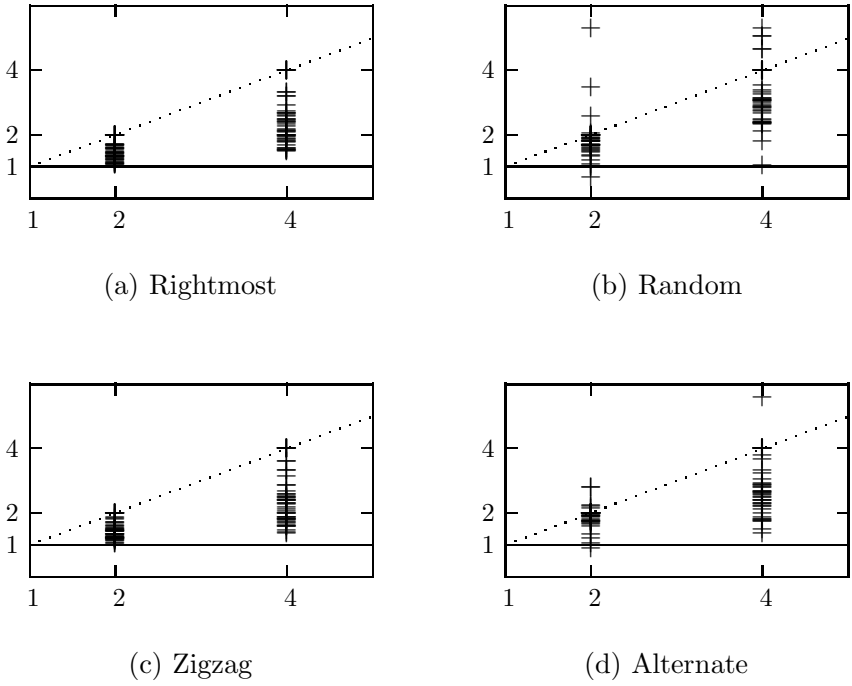

Fig. 14. Speedups for the different approaches.

the tool interaction protocol and is used to generate the code responsible for the communication and synchronization, as well as the interfaces that the tools shall implement to participate in the interaction. Also, we developed different approaches that impact on how the workload is distributed. We validated our approach via a series of experiments that show, first, that the distributed version does indeed achieve an interesting speedup in average and, second, that the so-called Random assignment choice approach is the most efficient policy.

In the future, we plan to use the interaction protocol as a basis to a grid-based approach to lazy theorem proving. We also envision to extend or adapt the proposed protocol so that propositional reasoning can be handled with a SAT-solver, based on some existing framework for propositional logic [7,16].

References

[1] Ball, T., B. Cook, S. K. Lahriri, and L. Zhang, *Zapato: Automatic theorem proving for predicate abstraction refinement*, in: *Proceedings of the 16th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science **3114** (2004), pp. 457–461.

[2] Ball, T., A. Podelski and S. 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*, Lecture Notes in Computer Science **2280** (2002), pp. 158–172.

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

[4] Bergstra, J. A. and P. Klint, *The discrete time ToolBus — a software coordination architecture*, Science

of Computer Programming **31** (1998), pp. 205–229.
URL citeseer.ist.psu.edu/bergstra98discrete.html

[5] Bryant, R., *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers **C-38** (1986), pp. 677–691.

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

[7] Chrabakh, W. and R. Wolski, *GridSAT: A Chaff-based distributed sat solver for the Grid*, in: *Proc. of the 2003 ACM/IEEE Conference on Supercomputing*, 2003, p. 37.

[8] Clarke, E., O. Grumberg, S. Jha, Y. Lu and H. Veith, *Counterexample-guided abstraction refinement for symbolic model checking*, Journal of the ACM **50** (2003), pp. 752–794.

[9] Couchot, J.-F., D. Déharbe, A. Giorgetti and S. Ranise, *Scalable automated proving and debugging of set-based specifications*, Journal of the Brazilian Computer Society **9** (2004), pp. 137–151.

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

[11] Déharbe, D., A. Imine and S. 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 (2004), pp. 271–275.

[12] Déharbe, D. and S. 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)* (2003), pp. 220–228.

[13] Déharbe, D. and S. 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.

[14] Detlefs, D. L., K. R. M. Leino, G. Nelson and J. B. Saxe, *Extended static checking*, Research Report 159, Compaq Systems Research Center (1998).

[15] Flanagan, C., R. Joshi, X. Ou, and J. B. Saxe, *Theorem proving using lazy proof explanation*, in: *Proceedings of the 15th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science **2725** (2003), pp. 355–367.

[16] Ganai, M. K., A. Gupta, Z. Yang and P. Ashar, *Efficient distributed sat and sat-based distributed bounded model checking*, in: *Correct Hardware Design and Verification Methods (CHARME 2003)*, Lecture Notes in Computer Science **2860**, 2003, pp. 334–347.

[17] Ganzinger, H., G. Hagen, R. Nieuwenhuis, A. Oliveras and C. 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)*, Lecture Notes in Computer Science **3114** (2004), pp. 175–188.
URL ftp://ftp.cs.uiowa.edu/pub/tinelli/papers/GanHNOT-CAV-04.pdf

[18] Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang and S. Malik, *Chaff: Engineering an efficient sat solver*, in: *Proc. of the 39th Design Automation Conference (DAC 2001)*, 2001.

[19] Nelson, G. and D. C. Oppen, *Simplification by cooperating decision procedures*, ACM Transactions on Programming Languages and Systems **1** (1979), pp. 245–257.

[20] Ranise, S. and D. Déharbe, *Applying light-weight 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).

[21] Ranise, S., 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)*, Lecture Notes in Computer Science **3405** (2004), pp. 372–386.

[22] Schulz, S., *E—a brainiac theorem prover*, Journal of AI Communications **15** (2002), pp. 111–126.