Verified Incremental Development of Lock-Free Algorithms

David Déharbe*, Loïc Fejoz†, Pascal Fontaine‡, Stephan Merz†
*Universidade Federal do Rio Grande do Norte, Natal, RN, Brazil
†LORIA–INRIA, Nancy, France
‡LORIA–Nancy University, Nancy, France
david@dimap.ufrn.br, {Loic.Fejoz,Pascal.Fontaine,Stephan.Merz}@loria.fr

***Abstract*—Mutual exclusion locks have traditionally been used for synchronizing the access of multiple processes to shared data structures. Multi-processor architectures offer new synchronization primitives that have given rise to algorithms for optimistic concurrency and non-blocking implementations. These algorithms are quite subtle and raise the need for formal verification. We propose a refinement-based method for designing and verifying non-blocking, and in particular lock-free, algorithms. We achieve good success in automatically discharging the verification conditions generated by this method, relying on first-order and SMT provers, augmented by a simple but useful instantiation heuristic. We present our method using a non-trivial running example due to Harris et al.**

I. Overview

When multiple processes access a shared data structure, some form of discipline is necessary to prevent conflicting accesses and potential inconsistencies. Traditionally, mutual exclusion via locking has been the main inter-process synchronisation abstraction. Although conceptually simple, this technique has a number of problems. Indeed, coarse-grained locking, by which a process gains exclusive access to the entire data structure, is unnecessary when processes wish to access different parts of the data and thus degrades overall performance. Fine-grained locking, on the other hand, is prone to deadlocks and priority inversions, giving rise to errors that are difficult to reproduce and hard to debug. Globally speaking, lock-based implementations are examples of a pessimistic approach to concurrency where the designer guards in advance against possible errors.

In contrast, optimistic concurrency control allows processes to execute as if they were working in isolation. When concurrent access is detected, process react appropriately in order to ensure overall correctness. This approach has been proposed since at least the early 1990s [8], [9]; it has received new attention with the advent of highly concurrent processor and system architectures. The corresponding algorithms rely on atomic processor instructions such as Compare-And-Swap (CAS), Load-Link/Store-Conditional (LL/SC) or Test-And-Set (TAS). It has been shown [16] that their use can make algorithms scale better. Lock-free algorithms avoid deadlocks by construction, but still place the burden of establishing overall correctness to shared data structures on the programmer. Software Transactional Memory (STM [20]) is a higher-level abstraction that does not explicitly require locks. It provides a simple interface to the application programmer who can pretend to write sequential code. Implementations of STM make use of non-blocking algorithms.

Because of their importance and intricacy, we believe that non-blocking algorithms are a prime target for techniques of verification and formal methods of system development. Traditionally, algorithm designers provide informal correctness arguments, and it is easy to overlook corner cases. Because arbitrarily many instances of the algorithms are expected to cooperate properly, standard finite-state model checking cannot by itself ensure overall correctness. Harris et al. [7] propose a clever abstraction technique, but still do not verify full correctness of their algorithms.

Linearizability [9] is the most widely accepted correctness property that non-blocking algorithms should satisfy. It requires that the externally observable behavior of the implementation corresponds to the outcome produced by some interleaving of atomically executed operations. From the user's point of view, the implementation can therefore be understood in terms of a sequential specification where each operation is described in isolation. Linearizability ensures that all invariants guaranteed by the atomic specification are preserved by the implementation. Formally, proving linearizability amounts to proving a refinement relationship between the non-atomic implementation and the atomic specification. Proposed techniques for verifying linearizability include

The work of the second author was supported by Microsoft Research through its European PhD Scholarship Programme.

```
word_t RDCSS(RDCSSDescriptor_t *d) {
  word_t r;
  do {
    r = CAS1(d->a2, d->o2, d);
    if (IsDescriptor(r)) Complete(r);
  } while (IsDescriptor(r));
  if (r == d->o2) Complete(d);
  return r;
}
```

```
word_t RDCSSRead(word_t *addr) {
  word_t r;
  do {
    r = *addr;
    if (IsDescriptor(r)) Complete(r);
  } while (IsDescriptor(r));
  return r;
}
```

```
void Complete(RDCSSDescriptor_t *d) {
  word_t v = *(d->a1);
  if (v == d->o1) {
    CAS1(d->a2,d,d->n2);
  } else {
    CAS1(d->a2,d,d->o2);
  }
}
```

Fig. 1. Implementation of the RDCSS operation.

types for atomicity [6] and reduction rules in the sense of Lipton [14]. Unfortunately, we have not been able to apply them for advanced algorithms such as those based on descriptors (see Sect. II), because the primitive operations do not commute. Methods based on separation logic such as [18] appear more promising, but their application currently appears to be limited to relatively simple data structures such as lists. Moreover, they currently lack support by powerful automatic provers. In principle, one could apply general-purpose formal methods such as assumption-commitment reasoning [10] or methods based on refinement such as B [1] or TLA^+ [13]. However, initial experience indicated to us that their use quickly leads to messy encodings and complicated proof obligations that can be difficult to understand and to prove. This motivated us to develop a custom framework for proving the correctness of such algorithms, optimized for typical techniques that appear in non-blocking algorithms and for support by automatic proof tools. In particular, we have extended our SMT solver by a simple instantiation heuristic that gives good coverage for the case studies that we have considered.

This paper is organized as follows: Sect. II gives a short overview of non-blocking algorithms and introduces the running example used in the remainder of the paper. Section III describes our method for modeling lock-free algorithms and explains the associated verification conditions for proving linearizability. Section IV describes our experiences with applying automatic provers and presents the instantiation heuristic that we have implemented for our SMT solver. Section V concludes the paper.

II. Non-Blocking Algorithms

Non-blocking algorithms mediate the (uniform) access of several processes to data structures held in shared memory. We assume here a *sequentially consistent* memory model [12]. In principle, non-blocking algorithms could be based on atomic read and write instructions, but this would be quite complex and inefficient. Modern processors provide instructions that provide somewhat more complex atomic memory operations. In particular, single-word CAS (CAS_1) is available on many processor architectures and has been shown to be universal [8] in the sense that it can be used to implement similar atomic operations, including multi-word CAS [7]. The effect of CAS_1 is to atomically execute the following code:

```
word_t CAS1(word_t *a, word_t o, word_t n) {
  word_t r = *a;
  if (r == o) { *a = n; }
  return r;
}
```

In particular, notice that CAS_1 never blocks and always terminates. Therefore, CAS_1 and similar primitive operations avoid deadlocks and priority inversion problems, although they can cause contention. Harris et al. [7] propose an implementation of multi-word CAS based on the following intermediate operation RDCSS:

```
word_t RDCSS(word_t *a1, word_t o1,
             word_t *a2, word_t o2, word_t n2) {
  word_t r =*a2;
  if ((r == o2) && (*a1 == o1)) { *a2 = n2; }
  return r;
}
```

RDCSS atomically updates the content of variable a2 if the values of a2 *and of another variable* a1 match specified values o2 and o1. RDCSS is not available as a primitive operation, and [7] proposes an implementation based on CAS_1 that appears in Fig. 1. Instead of passing multiple arguments, the implementation uses a single argument, a *descriptor*, that contains a field for every argument of the original operation. It is assumed that memory locations can store descriptors as well as ordinary values, and that descriptors can be distinguished from ordinary values. Processes use fresh descriptors for every invocation of the RDCSS operation. Most importantly, the implementation requires that memory locations that may be updated through the RDCSS operation be only read through the auxiliary RDCSSRead operation. Observe that RDCSSRead always returns an ordinary value and never a descriptor.

The RDCSS implementation begins by installing the descriptor at location a2 using CAS_1. If this succeeds, the variable indeed contained the expected value o2, and

the RDCSS operation is completed by reading the value of a1 and exchanging the descriptor against either n2 or o2, depending on whether a1 contained the expected value o1 or not. If another process concurrently accesses a2, the initial CAS_1 operation returns a descriptor. In this case, the implementation could simply back off, waiting for the first operation to complete. As an optimization, it instead completes the operation on behalf of the competing process; this is possible because the descriptor contains all the necessary information. When the first process later tries to complete its operation, it will no longer find its descriptor, and the final CAS_1 operation has no effect. Harris et al. [7] go on to similarly implement the CAS_n instruction using the RDCSS operation.

Our objective is to provide a method and tool support for proving the correctness of implementations such as RDCSS with respect to linearizability. That is, any multi-threaded execution of these operations appears as a possible execution of the same code where calls to RDCSS are replaced by atomic executions of the original RDCSS instruction, and calls to RDCSSRead are replaced by atomic reads of the variable.

III. Modeling and Refining Lock-Free Algorithms

We first describe our proposal for specifying (systems of) non-blocking algorithms and then introduce verification conditions for proving that a lower-level specification correctly refines a higher-level description of the same algorithms. Our method has been formalized and verified in the interactive proof assistant Isabelle/HOL [17].

A. Algorithm Specifications

Our method is intended for the simultaneous development of a finite fixed set $Alg_1, \ldots, Alg_m$ of algorithms. These access a (sequentially consistent) shared memory, represented by a set GV of global variables. They also access local variables LV_i; we write $PV_i = GV \cup LV_i$ for the variable signature of algorithm Alg_i. An *algorithm specification* (over set of variables PV) is given as an extended finite transition system: a finite-state control flow graph whose edges are labeled with transition predicates. These are formulas containing variables v and v', for $v \in PV$, that constrain the variable update during a transition (with the usual convention that v denotes the value before and v', the value after the transition). For example, the representation of the atomic RDCSS operation appears at the top of Fig. 2. Nodes q are labeled with two predicates: the state predicate $I(q)$ represents an invariant that should hold whenever the algorithm is at place q; it does not contain primed variables. The transition predicate $R(q)$, whose use is inspired by assume-guarantee methods of program verification, constrains the allowed transitions of other processes that coexist in the same system.[1] In order to be well-formed, an algorithm specification must satisfy the conditions

$$\begin{array}{rcl} I(q) \wedge \delta(q, q') & \Rightarrow & I'(q') \\ I(q) \wedge R(q) & \Rightarrow & I'(q) \end{array} \tag{1}$$

for all places q, q' (where $\delta(q, q')$ denotes the transition predicate associated with the transition from q to q' and $I'(q')$ denotes the invariant associated with place q' where all variables have been replaced by their primed copies): algorithm transitions must establish the invariant of the target state, while transitions of other processes should preserve it.

A *system specification* is just a set of algorithm specifications. It is well-formed if every algorithm specification is well-formed and, moreover, all (applicable) transitions of each algorithm Alg_i respect the rely predicates of all algorithms of the specification—including those of Alg_i itself, because several instances of the same algorithm may be executed in parallel by different processes. More formally, for any two algorithm specifications Alg_i and Alg_j over variables PV_i and PV_j such that $PV_i \cap PV_j = GV$,[2] we must have

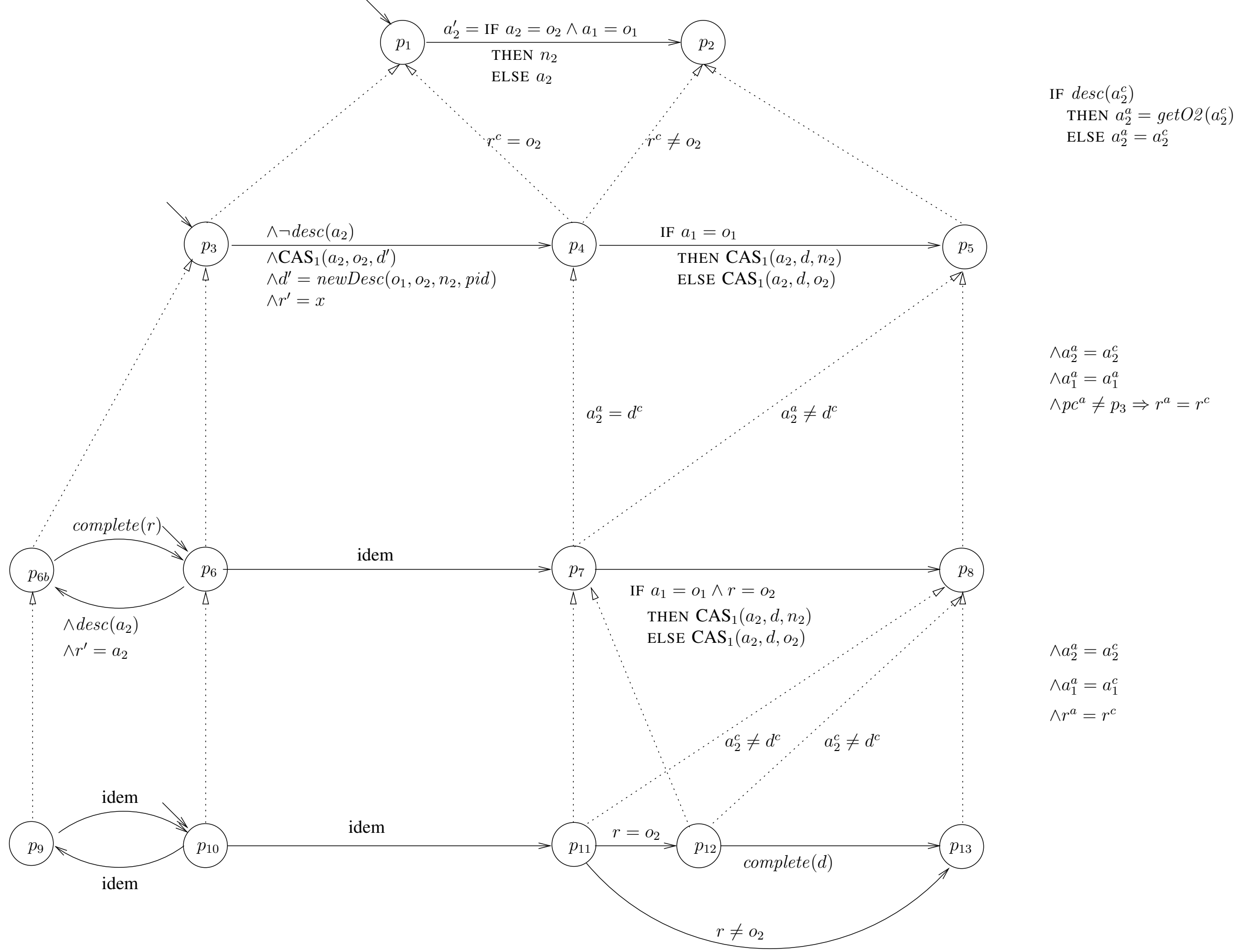
$$\begin{array}{l} I_i(q_i) \wedge I_j(q_j) \wedge \delta_i(q_i, q_i') \wedge LV_j{}' = LV_j \\ \quad \Rightarrow \ R_j(q_j) \end{array} \tag{2}$$

for all places q_i, q_i' of Alg_i and q_j of Alg_j.

At run-time, we model a *system* as consisting of an at most denumerable number of processes $Proc_1, Proc_2, \ldots$, where process $Proc_i$ is assumed to execute algorithm $Alg_{\alpha(i)}$, over a private copy of the local variables defined by that algorithm. The state of a system is thus given by a global environment $genv$ that assigns values to the variables in GV and a local state $(q_i, lenv_i)$ for each process $Proc_i$ such that q_i is a place of the algorithm specification $Alg_{\alpha(i)}$ and $lenv_i$ assigns values to a private copy of the variables $LV_{\alpha(i)}$. We write $penv_i$ for the joint valuation $(genv, lenv_i)$. A *system run* is an ω-sequence $\sigma = s^0 s^1 \ldots$ of system states such that the places of all processes at s^0 are the initial places of the corresponding algorithms, the process environments $penv_i^0$ at s^0 satisfy the invariants associated with these places, and every transition (s^k, s^{k+1}) corresponds to the transition of some process $Proc_i$ according to $\delta_{\alpha(i)}$, leaving all other local states unchanged, or to a stuttering

[1]These predicates are all trivial (TRUE) for the atomic RDCSS example.

[2]In particular, for $i = j$, we rename the local variables of one copy of the algorithm.



$$\text{CAS}_1(a_1, o_1, n_1) \triangleq \text{IF } a_1 = o_1 \text{ THEN } a_1' = n_1 \text{ ELSE } a_1' = a_1$$
$$complete(v) \triangleq \text{IF } a_1 = getO_1(v) \text{ THEN } \text{CAS}_1(a_2, v, getN_2(v)) \text{ ELSE } \text{CAS}_1(a_2, v, getN_2(v))$$
$$I(p_2) = I(p_5) = I(p_8) = I(p_{13}) \triangleq a_2 \neq d$$
$$I(p_4) \triangleq ((a_2 = d \wedge r = o_2) \vee (a_2 \neq d \wedge r \neq o_2))$$
$$R(p_2) = R(p_5) = R(p_8) = R(p_{13}) \triangleq a_2' \neq d$$
$$R(p_4) \triangleq \text{IF } a_2 = d \text{ THEN } a_2' = a_2 \text{ ELSE } a_2' \neq d$$
$$R(p_7) = R(p_{11}) \triangleq a_2' = a_2 \vee (a_2 = d \wedge complete(a_2))$$
$$I(p_7) = I(p_{11}) \triangleq a_2 = d \Rightarrow r = o2$$
$$I(p_{10}) = I(p_6) = I(p_{6b}) \triangleq desc(a_2) \Rightarrow getPid(a_2) \neq pid$$

Fig. 2. Stepwise development of the RDCSS implementation.

step, i.e. $s^{k+1} = s^k$. Stuttering steps are allowed in executions because lower-level refinements will typically add steps that map to stuttering at the abstract level of description.

The well-formedness conditions (1) and (2) ensure that all local invariants hold throughout any system run: for any system state s^k, every local state $penv_i^k$ satisfies the invariant $I_{\alpha(i)}(q_i^k)$ associated to the current place of the algorithm.

B. System Refinement

During system development, we wish to replace a high-level system specification by a more detailed one. We thus have to compare system specifications $Spec^a = \{Alg_1^a, \ldots, Alg_m^a\}$ and $Spec^c = \{Alg_1^c, \ldots, Alg_m^c\}$ at "abstract" and "concrete" levels of description. Whereas the set of algorithms is fixed, their specifications change, and we assume w.l.o.g. that these specifications are written in terms of distinct sets of variables. We expect the

Assume given two specifications $Spec^c = \{Alg_1^c, \ldots, Alg_m^c\}$ and $Spec^a = \{Alg_1^a, \ldots, Alg_m^a\}$, refinement relations $Ref = (Ref_1, \ldots, Ref_m)$, and gluing invariants $G = (G_1, \ldots, G_m)$. We say that $Spec^c$ refines $Spec^a$ modulo Ref and G if all of the following conditions hold:

- For every concrete initial state there exists an abstract initial state satisfying the gluing invariants:

$$\left(\bigwedge_{i=1}^{m} I_i^c(q_{0,i}^c)\right) \Rightarrow \exists GV^a, LV_1^a, \ldots, LV_m^a : \bigwedge_{i=1}^{m} G_i(q_{0,i}^c, q_{0,i}^a) \wedge I_i^a(q_{0,i}^a)$$

- Every concrete-level step corresponds to the abstract step of some algorithm, or to stuttering: for all $i, k = 1, \ldots, m$ and all places $(q_k^c, q_k^a) \in Ref_k$ the following implication is valid:

$$\begin{aligned}
&\delta_i^c(q_i^c, q_i^{c\prime}) \wedge \bigwedge_{k \neq i} q_k^{c\prime} = q_k^c \wedge LV_k^{c\prime} = LV_k^c \\
&\quad \wedge \bigwedge_{k=1}^{m} G_k(q_k^c, q_k^a) \wedge I_k^c(q_k^c) \wedge I_k^a(q_k^a) \\
&\Rightarrow \exists GV^{a\prime}, LV_1^{a\prime}, \ldots, LV_m^{a\prime} : \\
&\qquad GV^{a\prime} = GV^a \wedge \bigwedge_{k=1}^{m} LV_k^{a\prime} = LV_k^a \wedge G_k'(q_k^{c\prime}, q_k^a) \\
&\qquad \vee \bigvee_{j=1}^{m} \bigvee_{\{q_j^{a\prime} : (q_j^{c\prime}, q_j^{a\prime}) \in Ref_j\}} \delta_j^a(q_j^a, q_j^{a\prime}) \wedge G_j'(q_j^{c\prime}, q_j^{a\prime}) \\
&\qquad\qquad \wedge \bigwedge_{k \neq j} LV_k^{a\prime} = LV_k^a \wedge G_k'(q_k^{c\prime}, q_k^a)
\end{aligned}$$

Fig. 3. Proof obligations for refinement.

developer, besides writing the algorithm specifications, to link them by defining a *gluing invariant* relating the state representations at the two levels of description. Formally, the gluing invariant is expressed as a state formula $G_i(q^c, q^a)$ over the variables $PV_i{}^c \cup PV_i{}^a$, associated with the places of the concrete and abstract specifications.

Fig. 2 shows three levels of refinement for the RDCSS algorithm. The most detailed specification corresponds to the structure of the code shown in Sect. I. The gluing invariants for states linked by dotted arrows are shown in the right-hand column of the diagram, to be completed by the formulas that appear alongside the dotted arrows. For places not linked by dotted arrows, the gluing invariant is FALSE. To make the figure more readable, some annotations are omitted: the invariants and rely predicates assert that a_1, o_1, o_2, and n_2 are not descriptors, that d is a descriptor whose fields have the expected values and that are not descriptors themselves. As for the transition annotations, all variables that do not appear primed are implicitly left unchanged by the transition.

In order to justify that a (well-formed) specification refines another one, two verification conditions have to be proved:

1) for every concrete initial state satisfying the invariant there exists an abstract initial state that satisfies the abstract invariant and the gluing invariant, and
2) every concrete-level step corresponds to the abstract step of some algorithm, or to stuttering.

Although the basic idea of these conditions is common to formalisms based on refinement [1], [2], [11], [15], the precise formulation of the second condition should allow a concrete algorithm Alg_i^c to perform an abstract-level transition on behalf of some other abstract algorithm Alg_j^a, or a different instance of the same algorithm. In our running example, such a "non-local" refinement appears, among others, at place p_7 when another process removes the descriptor, implying $a_2^c \neq d$; hence the abstract algorithm moves to place p_5 whereas the concrete one remains at p_7. The precise verification conditions appear in Fig. 3.

Our method has been fully formalized in the interactive proof assistant Isabelle/HOL [17]. In particular, we have proved that if a concrete specification $Spec^c$ refines an abstract specification $Spec^a$ then for every run of a system based on $Spec^c$ there exists a run of the analogous system based on $Spec^a$ such that the system and gluing invariants are verified at every state. In particular, if the high-level specification is atomic, this implies that the implementation is linearizable.

```
01 function eq_instantiations(φ : formula, p : {+, −}) returns set of 𝒮
02     if φ is v = t and p is + or φ is v ≠ t and p is −, with v ∈ 𝒱, t ∈ 𝒯 then
03         return {{(v, t)}}
04     if φ is ¬ψ then
05         return eq_instantiations(ψ, inv(p))
06     if φ is ψ ∧ ψ′ then
07         let S = eq_instantiations(ψ, p) and S = eq_instantiations(ψ′, p) in
08             return {s ∪ s′ | s ∈ S ∧ s′ ∈ S′}
09     if φ is ψ ∨ ψ′ then
10         let S = eq_instantiations(ψ, p) and S = eq_instantiations(ψ′, p) in
11             return S ∪ S′
12     if φ is ψ ⇒ ψ′ then
13         let S = eq_instantiations(ψ, inv(p)) and S = eq_instantiations(ψ′, p) in
14             return S ∪ S′
15     return {}
16 end eq_instantiations
```

Fig. 4. Generating interesting sets of instantiations.

IV. Effective Tool Support

A. Generating Verification Conditions

We have implemented a verification condition generator that outputs the proof obligations corresponding to the well-formed conditions (1) and (2) of Sect. III-A and the refinement conditions of Sect. III-B in the format of different automatic and interactive theorem provers. These proof obligations can be quite large quantified first-order formulas (a few dozen lines per proof obligation for the RDCSS example), and discharging them interactively becomes quite tedious. The precise nature of the proof obligations and the background theory they should be evaluated in depends on the algorithm at hand. In the case of RDCSS, they only contain uninterpreted function symbols, and it is natural to consider the use of resolution-based theorem provers such as the E Prover [19] or Spass [22].

SMT (satisfiability modulo theories) solvers [3], [4], [5] are another breed of automatic deduction tools. They are able to handle large quantifier-free formulas that fall into decidable logical theories, combining state-of-the-art SAT solver techniques to efficiently handle the Boolean structure of the formula with dedicated decision procedures for each logical theory. Unfortunately, we cannot use them directly because our proof obligations contain (existential) quantification corresponding in particular to the choice of an abstract successor state. On manual inspection of the generated formulas, it usually appears to be quite straightforward to find the required instantiations. However, when we applied standard SMT solvers such as CVC3 or Z3 to our running example, their instantiation heuristics were able to prove fewer verification conditions than the resolution-based theorem provers we tried. We now discuss a simple heuristic for instantiating quantifiers that we have implemented within our SMT solver VERIT (a successor HARVEY [5]), and that we have found very useful for discharging the proof obligations that are generated by the method of Sect. III.

B. An Equality-Driven Instantiation Heuristic

Many quantified formulas generated by the refinement method of Sect. III-B contain subformulas similar to

$$\exists v_1, v_2 : (v_1 = t_1 \land v_2 = t_2) \lor (v_1 = u_1 \land v_2 = u_2) \\ \land\ \psi(v_1, v_2)$$

for ground terms t_1, t_2, u_1, u_2. This syntactic structure provides a hint that the instances $\psi(t_1, t_2)$ and $\psi(u_1, u_2)$ are relevant for the proof.

In order take into account this hint, the prover needs

- a heuristic to derive such instantiations of quantified formulas;
- to be able to produce and handle lemmas such as $\varphi(t) \Rightarrow (\exists x\, \varphi(x))$ and $(\forall x\, \varphi(x)) \Rightarrow \varphi(t)$. In the case of an incremental SMT solver, these lemmas may be produced as requested and added conjunctively to the original formula, whenever the SMT solver is unable to prove unsatisfiability from the already known instances.

Note that our proof obligations typically contain large quantified formulas[3]. In order to be able to scale up to

[3]Quantified subformulas may contain more than 1000 symbols, excluding separators.

	Proved	Resigned	Timeout
CVC3	116	0	25
Z3	123	18	0
Spass 3.0	95	—	2
E-prover	122	—	19

Fig. 5. Performances of some competing provers on the RDCSS proof obligations.

the size of these formulas, the heuristics should avoid useless instantiations such as $\{v_1 \leftarrow t_1, v_2 \leftarrow t'_2\}$ and $\{v_1 \leftarrow t'_1, v_2 \leftarrow t_2\}$ in the above example.

Let $\mathcal{V}$ denote the set of quantified variables, $\mathcal{T}$ the set of ground terms. A variable instantiation is a pair in $\mathcal{V} \times \mathcal{T}$. A set of variable instantiations is a called a multi-variable instantiation. We denote $\mathcal{S} = \mathbb{P}(\mathcal{V} \times \mathcal{T})$ the set of multi-variable instantiations. Note that this definition makes it possible that a multi-variable instantiation contain two pairs with the same variable. So, let $S \in \mathcal{S}$ be a multi-variable instantiation, $S^{\downarrow}$ is a multi-variable instantiation such that:

1) $S^{\downarrow}$ is a subset of S;
2) $S^{\downarrow}$ and S have the same domain;
3) $S^{\downarrow} \in \mathcal{V} \nrightarrow \mathcal{T}$ is a partial function from $\mathcal{V}$ to $\mathcal{T}$;

Note that, for a given S, several sets may satisfy the conditions enumerated above. For example, $\{(v_1, t_1), (v_1, t'_1)\}^{\downarrow}$ may be either $\{(v_1, t_1)\}$ or $\{(v_1, t'_1)\}$.

The main function for our heuristic derives a set of multi-variable instantiations from a given formula; it appears in Fig. 4 (the second parameter $p \in \{+, -\}$ records the polarity of the formula). This algorithm recurses over the structure of the formula and combines the result obtained from the sub-formulas according to the current boolean connector. When the connector is conjunction (lines 06–08), then the result is the set of unions of the multi-instantiations from each subformulas. When the connector is disjunction (lines 09–11), the result is the union of the sets of multi-instantiations. Negation and implication are handled by adapting the polarity of the subformulas (lines 04–05 and 12–14). The first base case (lines 02–03) corresponds to equalities between quantified variables and ground terms, and the second base case correspond to formulas that do not match the previously enumerated patterns. In the former case, the result is a set with a single multi-instantiation composed of a single instantiation. In the latter case the result is the empty set.

The heuristics instantiates a quantified subformula $\mathrm{Q}\mathcal{V}\,\varphi(\mathcal{V})$ (where $\mathrm{Q} \in \{\forall, \exists\}$ and $\mathcal{V}$ is a list of variables) by first computing the set of multi-variable instantiations $E(\varphi) = \mathit{eq_instantiations}(\varphi, +)$ and then instantiating φ with $S^{\downarrow}$, for each S in $E(\varphi)$.

After we enabled this heuristic for VERIT, it was indeed able to discharge all verification conditions for the RDCSS example, with a time bound of 5 seconds per proof obligation. As reported in Fig. 5, we ran several different automatic provers on the same proof obligations. The SMT solvers CVC3 and Z3[4] failed to prove around 20 formulas each. The E prover yielded a similar ratio of unproved formulas. Spass failed for 44 formulas due to some resource exhaustion. On other examples where this problem did not occur, Spass slightly outperformed the E prover. We used a 5 seconds time limit, on an Intel® Core™2 CPU at 2.16GHz with 2 Gb RAM running linux 2.6.24. The same experiments were conducted with a 30 seconds time limit without significant changes: only Spass proved one more formula. One should however note that our comparison is preliminary and rather biased. In particular, all provers except VERIT were run with their default parameters.

V. Contribution and Outlook

We have proposed a refinement-based method for the deductive verification of lock-free algorithms. Our notion of refinement reflects linearizability and accomodates non-atomic implementations modulo data and process refinement. It has been illustrated over a non-atomic implementation of the RDCSS operation, which is representative of the kind of operations used in modern multi- and many-core environments.

Our method has been formalized and verified in Isabelle/HOL. Besides giving us full confidence in the correctness of the method, this formalization has helped us to detect some subtle conditions that could have been overlooked in a pencil-and-paper proof, and to test some alternative definitions during the design of the method. A verification condition generator supports applications of the method and outputs proof obligations in the formats of SMT solvers and automatic theorem provers for first-order logics. We have presented a heuristic for quantifier reasoning that we have implemented in our SMT solver,

[4]The executables of those provers are the ones used in the 2008 edition of the SMT-COMP competition for SMT solvers.

enabling us to prove all or almost all proof obligations automatically for our running example. In general, the generated formulas do not fall into decidable first-order theories, so we cannot hope for full automation in all cases.

We intend to validate the method further by using it to verify implementations of non-blocking data structures such as those that appear in [16], [21]. We are very interested in studying liveness properties of such algorithms. It will be more challenging to relax the underlying assumption of a sequentially consistent memory model. Verifying higher-level abstractions such as software transactional memory [20] will also be of great interest.

With respect to the interaction with our SMT solver, we are working on two improvements. First, proof attempts frequently fail initially when developing the models, and it would be helpful to improve the output of counter-models provided by the automatic deduction tools by presenting them in terms of the algorithm specification. Second, we have made our SMT solver proof-producing, and we would like to certify the proofs using a trusted verifier such as the kernel of Isabelle.

REFERENCES

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] R. Back and J. von Wright. *Refinement calculus—A systematic introduction*. Springer Verlag, 1998.

[3] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[4] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[5] D. Déharbe, P. Fontaine, S. Ranise, and C. Ringeissen. Decision procedures for the formal analysis of software. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *Intl. Coll. Theor. Aspects Comp. (ICTAC)*, volume 4281 of *Lecture Notes in Computer Science*, pages 366–370, Tunis, Tunisia, 2007. Springer.

[6] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *ACM SIGPLAN Intl. Ws. Types in language design and implementation*, pages 47–58, Long Beach, CA, 2005. ACM.

[7] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In D. Malkhi, editor, *Proc. 16th Intl. Symp. Dist. Comp.*, volume 2508 of *Lecture Notes in Computer Science*, pages 265–279, Toulouse, France, 2002. Springer.

[8] M. P. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):124–149, 1991.

[9] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, 1990.

[10] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Prog. Lang. Syst.*, 5(4):596–619, Oct. 1983.

[11] B. Jonsson. Simulations between specifications of distributed systems. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91, 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 346–360, Amsterdam, The Netherlands, 1991. Springer.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, 28(9):690–691, 1979.

[13] L. Lamport. *Specifying Systems*. Addison-Wesley, Boston, Mass., 2002.

[14] R. J. Lipton. Reduction: a new method of proving properties of systems of processes. In *2nd ACM Symp. Princ. Prog. Lang. (POPL)*, pages 78–86, Palo Alto, CA, 1975. ACM.

[15] N. Lynch and F. Vaandrager. Forward and backward simulations. Part I: Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[16] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared-memory multiprocessors. *J. Par. Dist. Comp.*, 51(1):1–26, 1998.

[17] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer Verlag, 2002.

[18] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. *SIGPLAN Not.*, 42(1):297–302, 2007.

[19] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[20] N. Shavit and D. Touitou. Software transactional memory. In *Distributed Computing*, pages 204–213, 1995.

[21] H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Gothenburg University, 2004.

[22] C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System description: Spass version 3.0. In F. Pfenning, editor, *21st Intl. Conf. Automated Deduction (CADE 2007)*, volume 4603 of *Lecture Notes in Computer Science*, pages 514–520, Bremen, Germany, 2007. Springer.