

# Towards Automatic Stability Analysis for Rely-Guarantee Proofs

Hasan Amjad and Richard Bornat

Middlesex University School of Computing Science, London NW4 4BT, UK

Hasan.Amjad@cl.cam.ac.uk      R.Bornat@mdx.ac.uk

**Abstract.** The Rely-Guarantee approach is a well-known compositional method for proving Hoare logic properties of concurrent programs. In this approach, predicates in the proof must be proved invariant (or stable) under interference from the environment. We describe a framework, and a prototype implementation, for automatically detecting and repairing instability in such proofs. The method uses a combination of model checking, abstract interpretation, SMT and flow-control refinement.

## 1 Introduction

Multi-core and multi-processor computing systems are now mainstream. Hence, shared-memory concurrency, where multiple threads have read/write access to the same memory space, is becoming increasingly common. Consequently, concurrent programs are the focus of much recent research on automatically proving program properties. Often, the assertions we would like to prove are not amenable to existing automatic analyses. This paper studies one such scenario, and shows how existing automatic techniques can nonetheless help the proof process.

The main challenge in proving properties of concurrent programs, and indeed in their design, is dealing with interference, i.e., the possibility that threads may concurrently make changes to the same memory address. The concurrent programming community has evolved several synchronisation schemes to avoid interference. Most rely on some form of access denial, such as locks. Though locks make it easy to reason about correctness, they may also cause loss of efficiency as threads wait to acquire access to needed resources. Locking schemes have thus become increasingly fine-grained, attempting to deny access to the smallest possible size of resource, to minimise waiting and maximise concurrency. The ultimate form of such fine-grained concurrency are programs that manage without any synchronisation at all [17].

The finer the concurrency, the more involved the logic for avoiding interference. This logic must implicitly or explicitly take the actions of other threads into account. This is a problem for program proofs where we strive for modularity, i.e., we wish to be able to reason about a piece of code in isolation from the various environments in which it could execute.

Rely-guarantee (RG) reasoning [16] offers a fairly widely used solution to this problem within the framework of Hoare-style program proofs [14]. Broadly speaking, RG encodes the environment into the proof: all pre- and post-conditions of every Hoare triple must be shown to be unaffected (or stable) under the actions of the environment. Once non-interference has been established, the proof can be carried forward exactly as for a sequential program. Automatically ensuring such non-interference can be problematic in many cases.

In this paper, we describe preliminary progress on a possible solution, that relies critically on state-of-the-art tools from the model checking, abstract interpretation and Satisfiability-modulo-Theories (SMT) research communities. Our stability analysis engine can be integrated with a verification condition generator, allowing greater automation for Hoare logic proofs of concurrent programs.

The next section gives relevant background. We then describe our method, and give examples to illustrate shortcomings and possible developments. We assume some familiarity with program proofs in Hoare logic [14], and with model checking [7].

## 2 Preliminaries

### 2.1 Rely-guarantee Reasoning

Rely-guarantee (RG) is a compositional verification method for shared memory concurrency introduced by Jones [16]. Interference between threads is described using binary relations. In that treatment, post-conditions were relational, so assertions could talk about the state before and after an action. Here, in line with traditional Hoare logic, we shall use post-conditions of a single state, as this usually makes for simpler proofs. In either case, the essence of RG is unaffected by this choice.

Our command language will be the one used by Jones, i.e., with assignment, looping, branching, sequential composition and parallel composition, using C-like syntax. For parallel composition we assume standard interleaved execution semantics, i.e., the threads of a program have access to some shared state, and atomic instructions occur interleaved.

Program variables will range over  $\mathbb{N}$ . It may seem odd to have program variables range over infinite types. In practice however, reasoning about numbers with the aid of abstraction, has been found to be more tractable than reasoning about finite but huge state spaces over machine words or bit-vectors, which are harder to abstract due to overflow and underflow corner cases. We made this choice primarily for ease of tool use while developing our prototype.

RG can be seen as a compositional version of the Owicki-Gries method [18]. The specification for a command  $C$  is a four-tuple  $(P, R, G, Q)$ , where  $P$  and  $Q$  are the usual Hoare logic pre- and post-condition assertions on a single state, and  $R$  and  $G$  are binary relations on states (often called “two-store” predicates).

$R$  and  $G$  summarise the properties of the individual atomic actions invoked by the environment and the thread respectively. An action is given as a binary

relation on the shared state, and is written  $g \rightsquigarrow \bar{v} := \bar{e}$ . This notation indicates that, if the guard  $g$  is true, then the action simultaneously updates the vector of shared state variables  $\bar{v}$  pointwise with the value of the expression vector  $\bar{e}$ , and does nothing otherwise. Note that that update need not be atomic. For example, the action corresponding to the command  $\mathbf{x} = \mathbf{x} + 1$ , that increments a shared integer  $x$ , might be written as  $\mathbf{true} \rightsquigarrow x := x + 1$ . We will elide  $\mathbf{true}$  guards.

$C$  satisfies its specification,  $(P, R, G, Q) \models C$ , if from a state satisfying  $P$ , and under environmental interference at most  $R$  (the *rely*),  $C$  causes interference at most  $G$  (the *guarantee*), and if it terminates, it does so in a state satisfying  $Q$ .

$G$  is the relation given by the reflexive and transitive closure of all actions of the thread being specified.  $R$  is similarly the reflexive and transitive closure of all the actions of the environment, i.e., the actions comprising  $R$  are just the  $G$  actions of all the other threads. We overload the notation for  $R$  and  $G$  to indicate either relations or predicates, as convenient.

Taking reflexive and transitive closures has the effect of forgetting all control-flow information. This has the disadvantage of overapproximating the behaviours of the environment, but the advantage that any proof will hold for arbitrary numbers of threads of the proved code (new code may of course break the proof).

The actions are given by manual annotation, as in general, automatic action discovery is non-trivial. The reason is that actions can only mention shared state variables, whereas in the code shared state variables may often be assigned the value of an expression containing non-shared (or *local*) variables. Fully automatic action annotation would thus require an iterative computation that may end up exploring the full state space of the program, and this we wish to avoid. For the toy programs we have looked at so far, this has not been a problem. For very large programs, one can always annotate a procedure call with a single action, and not descend into the call itself.

An assertion  $P$  on a single state is considered *stable* under interference from a binary relation  $R$  if  $(P; R) \Rightarrow P$ , i.e., if  $P(s)$  and  $(s, s') \in R$ , then  $P(s')$ . More specifically, if  $P$  is the pre-condition for some command  $C$ , then it must continue to hold after any environment action, before the execution of  $C$ .

Jones gives a full proof system for the satisfaction relation, but we will not need it for this work. However, we reproduce the two critical rules here, to make our assumptions about RG concrete. The first rule is parallel composition, where  $\parallel$  is the interleaving parallel composition operator.

$$\frac{(P_1, R \vee G_2, G_1, Q_1) \models C_1 \quad (P_2, R \vee G_1, G_2, Q_2) \models C_2}{(P_1 \wedge P_2, R, G_1 \vee G_2, Q_1 \wedge Q_2) \models C_1 \parallel C_2} \quad (1)$$

For our purposes, we assume that a read or write of a single boolean or integer is the highest level of hardware atomicity, e.g., writing two booleans is not hardware atomic. This assumption is satisfied by most machine architectures. The second rule tells us what it means for a command to be atomic,

$$\frac{(P, \text{id}, G, Q) \models C \quad P \text{ stable under } R}{(P, R, G, Q) \models \text{atomic}(C)} \quad (2)$$

meaning that if a command satisfies a specification with an empty rely, and the pre-condition is stable under some rely  $R$ , then the command may be treated as if it executed atomically. Hardware atomic commands are of course trivially atomic.

Note one departure from standard RG: the post-condition of the very last line of code is not checked for stability. It is instantaneously true immediately after execution of that line. At this point, either the thread terminates, so that we do not care whether the environment interferes with the post-condition, or, the thread resumes execution from some command the pre-condition of which will be the same as this post-condition, and thus will be checked for stability.

## 2.2 Temporal Logic Model Checking

Let  $V$  be the set of program variables (or *state variables*) used in a program (with appropriate scope management, which we ignore without loss of generality). Each  $v \in V$  ranges over a non-empty set of values  $D_v$ . The state space  $S$  of the program is given by  $\prod_{v \in V} D_v$ . A single state of the program is then a value assignment to each  $v \in V$ .

Suppose  $AP$  is the set of all those atomic propositions over  $V$  that we might use in the specification of a program. Then we can turn the program into a state machine  $M$  represented as a tuple  $(S, S0, T, L)$  where  $S$  is the set of states,  $S0 \subseteq S$  is the set of initial states,  $T \subseteq S \times S$  is the transition relation, and  $L : S \rightarrow 2^{AP}$  labels each state with the subset of  $AP$  that is true in that state.

A temporal logic augments propositional logic with modal and fix-point operators. The semantics of a temporal logic formula in which the atomic propositions range over  $AP$  can be expressed in terms of sets of states and/or sequences of states of  $M$ . If we turn a program into a state machine, we can use temporal logics to express temporal properties of the program.

The most common such property is the global invariant, i.e., a property that holds in all reachable states of a state machine, or equivalently, always holds during the execution of a program. We denote this by

$$M \models \mathbf{G}f$$

where  $M$  is the state machine,  $f$  is the global invariant and  $\mathbf{G}$  is the ‘‘Globally’’ temporal logic operator. The semantics of  $\mathbf{G}f$  are simply that  $f$  must hold in every state of  $M$  that is reachable from the initial states of  $M$ .

$M \models \mathbf{G}f$  can be checked automatically using proof procedures known as model checkers, subject only to time and space constraints. More importantly, if the proof attempt fails, the model checker can return a counterexample, which is an execution path (sequence of transitions) leading from an initial state to a state in which the invariant is not satisfied. See [7] for details.

The problem of model checking global invariants is in general undecidable when the state space is infinite. However, the ability to produce counterexamples has led to the development of counterexample guided abstraction refinement (CEGAR) [8, 19], where the state space is first abstracted to a simpler one,

and if the constructed abstraction is too general it can often be automatically iteratively refined until the desired property is verified.

We do not need to describe model checking or CEGAR in more depth, particularly as there are many different abstraction schemes and CEGAR techniques. Further details may be found in [7, 8, 13, 19].

### 3 The Problem of Instability Detection and Repair

Rule (2) of RG is used to establish a command as atomic. The command is then free from interference, and can be reasoned about using standard Hoare logic rules for sequential programs. The key requirement is to prove that the pre-condition  $P$  of the command is stable under the rely  $R$ . This is a two-step process. First, one *detects* whether or not  $P$  is stable. Second, if  $P$  is unstable, one *repairs* it; the usual way is by weakening  $P$ . This stability analysis is thus the key to reducing proofs about concurrent programs to proofs about sequential programs.

Instability can be detected either manually, or sometimes automatically using syntactic checks [22]. Once detected, it is repaired by weakening  $P$ , either manually or using ad hoc abstraction heuristics if the underlying domain permits [6, 9].

As a simple example, consider the assertion  $x = 10$  that is unstable under the environment consisting of the single action  $x := x + 1$ . In general, detecting this automatically is not straightforward.

To repair this, a common approach is to disjunctively add state based on the action, starting with the states satisfying the assertion, i.e.,

$$\begin{aligned} &(x = 10) \\ &(x = 10 \vee x = 11) \\ &\vdots \end{aligned}$$

and hope to reach a fixpoint. Unfortunately, in this case such naive automatic stabilisation will not terminate. To repair manually, we use the boolean abstraction  $\alpha(x) \iff x \geq 10$ , where  $\alpha$  is a total *abstraction function* [7]. Under this abstraction the action above becomes the identity action (now on booleans) in all cases except when  $x = 9$ , but in that case  $x = 10$  does not hold anyway, so we have stability immediately, and the assertion is stabilised to  $x \geq 10$ .

Automatic instability detection and repair are thus both non-trivial problems. Further, we observe that weakening  $P$  is not the only way to achieve stability. From the definition of stability, strengthening the rely is also a possible solution. We shall present a method that combines both approaches.

We have seen in §2.2 that this problem of finding exactly the right level of abstraction also occurs in model checking. It is our hope that the model checking solution (i.e., CEGAR) can be applied to stability analysis as well. If so, the vast amount of model checking research on this topic can be brought to bear on the problem.

## 4 Automatic Stability Analysis

Our solution is to represent  $R$  and  $G$  as state machines  $M_R$  and  $M_G$  of actions. Stability detection for a predicate  $P$  is then reduced to model checking that  $P$  is a global invariant over  $M_R$ . If the check fails, we use the counterexample trace provided by the model checker to either strengthen  $R$ , or weaken  $P$ , and then repeat the check until success or irreparable failure. The process can diverge because over an infinite state space there may be an infinite sequence of successively weaker assertions, none of which is stable. For this reason, for now we always prefer strengthening. A further novelty of our approach is that we perform weakening only with respect to the part of the environment that is causing instability (rather than the full environment), thus reducing the risk that the program proof will fail because the stabilised precondition was too weak.

### 4.1 Detection

The state machine for the guarantee condition  $G_t$  for a thread  $t$ , is  $M_t = (S_t, S0_t, T_t, L_t)$  and is constructed as follows. Let  $V_t$  be the set of all shared program variables used in  $t$ . Then  $S_t$  is constructed like  $S$  in §2.2.  $S0_t$  is those states of  $S_t$  in which all  $v \in V_t$  are assigned their initial values if any. Let  $a_l$  be the (possibly empty) set of actions associated with the command on line  $l$  of the thread code (having a set of actions associated with a command allows us to abstract procedure calls, and also to use auxilliary actions without violating atomicity constraints). Then

$$T_t(s, s') = \bigvee_l \left( \bigwedge_{a \in a_l} a(s, s') \right)$$

Note that this transition relation is not tracking control-flow. Finally,  $L_t(s) = \{p \in AP \mid p(s) = \mathbf{true}\}$ .  $L_t$  is a technical requirement for defining the semantics of temporal logics; it gives the set of states in which a given atomic proposition  $p \in AP$  is true.

Now suppose we have the guarantee state machines for all threads of the program. To do stability detection for some assertion  $P$  in the proof of some thread, let the environment  $E$  be the set of all other threads (i.e., all threads less the one for which the proof is being done). Then we construct the state machine for  $R$ ,  $M_R = (S_R, S0_R, T_R, L_R)$  over variables  $V_R = \bigcup_{t \in E} V_t$ , as follows

$$\left( \prod_{v \in V_R} D_v, \prod_{v \in V_R} \mathit{init}(v), \biguplus_{t \in E} T_t, \lambda s. \{p \in AP \mid p(s) = \mathbf{true}\} \right)$$

where  $\mathit{init}(v)$  gives the set of possible initial values of  $v$ . Note that the transition relation  $T_R$  is simply a union of the transition relations of the environment's threads, so it is also not tracking control flow.

Concretely, the transition system of  $M_R$  non-deterministically chooses a thread  $t \in E$ , after which  $M_R$  behaves like  $M_t$  for the duration of the execution

of some action of  $t$ , after which it again picks a thread and repeats. Once we have  $M_R$ , the stability check is reduced to invoking a model checker to confirm that

$$M_R \models \mathbf{G}P$$

**Theorem 1.** *Ignoring valuations of  $P$  on unreachable states of  $M_R$ , we have*

$$M_R \models \mathbf{G}P \Rightarrow (P; R) \Rightarrow P$$

*Proof.* If  $M_R \models \mathbf{G}P$  then  $P$  holds in all reachable states of  $M_R$ . A state  $s$  is reachable iff for  $s_0 \in S_{0R}$   $R(s_0, s)$ . Now  $P$  holds in all states, so in particular in  $s_0 \in S_{0R}$ , and all  $s$  and  $s'$  such that  $R(s', s)$ . Since  $(P; R) \Rightarrow P$  is equivalent to  $\forall ss'. P(s) \wedge R(s, s') \Rightarrow P(s')$ , we have our result.  $\square$

Note that ignoring valuations of  $P$  on unreachable states of  $M_R$  is fine because we do not care whether  $P$  is stable or unstable for situations that can never happen. It is possible that  $P$  may be stable on all reachable states and unstable only in some unreachable states, in which case the model checker may still fail the stability check (since the reachable states of  $M_R$  over-approximate the reachable states of the environment). This is a completeness issue, and is dealt with by a refinement-based solution in §4.2.

So if  $P$  is a global invariant of  $M_R$  then  $P$  is stable under  $R$ . If not, the model checker will supply a counterexample, which will be a sequence of actions leading to a state in which  $P$  is not satisfied.

A global invariant is a safety property in the sense that a violation of the property is caused by the explicit occurrence of an observable event. We note here that rule (1) of RG, which appears to be performing circular reasoning, is sound for safety properties [1].

We use the software model checker BLAST [5], to check that  $M_R \models \mathbf{G}P$ . Forgetting control flow considerably simplifies  $M_R$ , easing the model checker’s task. Our prototype implementation generates a C program corresponding to  $M_R$ , annotated with a BLAST assertion corresponding to  $P$ . The non-deterministic thread selection is easily constructed from the BLAST boolean non-deterministic choice primitive. We denote by  $mc(M, P)$  a call to the model checker, that checks whether assertion  $P$  is a global invariant over state machine  $M$ . This returns either **true** or a counterexample trace  $\pi$  as a sequence of actions.

Indirectly, the stability check relies on the model checker’s underlying SMT solver’s ability to check assertions in combinations of various theories. Thus, we can do instability detection in any theory supported by BLAST (or some software model checker), and so are able to always use state-of-the-art software model checking techniques for free. At this point we are already ahead of the game by not being limited to requiring syntactic stability checks.

## 4.2 Repair

As mentioned earlier, the standard approach to instability repair is by weakening  $P$ , but an alternative is to strengthen  $R$ . The latter approach can be useful if

weakening  $P$  would make it impossible to prove the Hoare logic property we are interested in. Also, as noted in §4.2, too weak an  $R$  can lead to incompleteness in stability detection. We now present a method that combines both approaches.

In standard RG,  $R$  and  $G$  are the reflexive transitive closures of their constituent actions. This representation corresponds to state machines that can perform any action from any state. Indeed,  $M_R$  above is precisely this state machine (since the transition relation is not tracking control-flow).

The obvious way to strengthen  $R$  is to re-introduce control-flow information into  $M_R$ . If the stability of  $P$  depends on the sequencing of actions, then such a refinement of  $R$  will allow us to stabilise  $P$  without weakening it. We shall see an example of this later. On the other hand, by introducing control-flow in  $R$ , we begin losing scalability in the sense that the model checker's task becomes harder, and the overall proof need not hold for arbitrary numbers of threads of the same piece of code, but only for the number that is model-checked.

To do this strengthening, we need to be able to track control-flow in each thread. Four preliminary steps are required:

1. We add, for each action  $a$  of each  $t \in E$ , the set of all actions of  $t$  that may execute immediately after  $a$ . We call this the *successor set* of  $a$ , denoted by  $\text{succ}(a)$ .
2. We attach to each action  $a$  a boolean flag  $\text{pin}(a)$ . If  $\text{pin}(a)$  is true, then we say that  $a$  is *pinned*, i.e., intuitively, we are now tracking control flow for  $a$ .
3. For each thread  $t$ , we add a special dummy action  $\text{start}_t$ , which is never executed, but for which  $\text{succ}(\text{start}_t)$  is the set of all actions of  $t$  that can be the very first action executed when  $t$  starts.
4. We maintain a function  $\text{last}(t)$ , which returns the most recent action executed by  $t$ , or  $\text{start}_t$  if  $t$  has not executed any action yet.

Till now, after the non-deterministic choice of thread in  $T_R$ , the action to execute was chosen non-deterministically as well. Now, let  $\text{choose}(n)$  for  $n > 0$  be the function that non-deterministically chooses a number between 0 and  $n - 1$ , and let  $|t|$  be the number of distinct actions of a thread  $t$ . Suppose thread  $t$  is picked as the next thread to run. The choice of next action in  $M_R$  is then determined as follows:

```

procedure pick_action( $t$ )
1.  $\text{tmp} \leftarrow \text{choose}(|t|)$ 
2. if ( $\text{pin}(\text{tmp})$ ) then
3.   if  $\text{tmp} \notin \text{succ}(\text{last}(t))$  then goto 1
4. return  $\text{tmp}$ 

```

The intuition is that an unpinned action can execute at any time, whereas a pinned action is only allowed to execute if it is in the successor set of the most recently executed action of that thread. This allows fine-grained control over adding execution sequencing information to  $M_R$ . This code is never executed, only model-checked, so non-determinism and infinite loops are perfectly acceptable.

The rely control-flow refinement then works as follows (recall that  $mc$  is a call to the model checker, return either **true** or a counterexample trace  $\pi$ ):

```

procedure  $mcr(M_R, P)$ 
1.  $\forall a. pin(a) \leftarrow \mathbf{false}$ 
2. match  $mc(M_R, P)$ 
3. case true : return SUCCESS
4. case  $\pi$  : let  $\langle a_0, a_1, \dots, a_m \rangle = \pi$ 
5. if  $\forall a \in \pi. pin(a)$  then return FAIL
6.  $tmp \leftarrow a_i \in \pi$  s.t.  $\neg pin(a_i) \wedge \forall j > i. a_j \in \pi \Rightarrow pin(a_j)$ 
7.  $pin(tmp) \leftarrow \mathbf{true}$ 
8. goto 2

```

The rely refinement loop works by adding control-flow information for actions in the counterexample trace until either the model checker reports success, or returns a counterexample trace in which we are already tracking control-flow for all actions of the trace. In the latter case, it is clear that there is at least one execution sequence where stability fails even when nothing executes out of sequence.

We must ensure that rule (1) of RG is still sound since the rely may now not be the reflexive transitive closure of the environment's actions. The other rules are not affected. The key is to ensure that the guarantee of each thread is stronger than the rely of every other thread. This is indeed the case: since our refinement only re-introduces control-flow information, the refined state machine will never under-approximate the actual program, and so the guarantees will allow all valid program executions.

With  $mcr$ , we simply fail if further control-flow refinement of the rely cannot repair instability. We now add weakening of  $P$ . For this, we amend  $mcr$  to return  $\pi$  instead of FAIL in line 5. Further, we assume access to a procedure  $weaken(M)$  which infers invariants on variable occurrences at various control points in a given state machine, and returns the list of invariants thus found.

We use the INTERPROC tool [15] for this functionality, as it provides an easy to use interface to state-of-the-art abstract interpretation libraries for numerical domains. Since we are ignoring the heap for now, our case studies use mainly arithmetic and static arrays. The input of INTERPROC is a program in an academic imperative language that supports numerical types. The underlying engine of INTERPROC consists of a generalised fixpoint computation which in turn uses abstraction interpretation libraries such as APRON [3]. Since these work over numerical domains only, our current examples are also limited to assertions over number-valued variables.

The output is the program code annotated with invariants on program variable occurrences at various control points in the code. Invariants at the same control point hold simultaneously and can be conjoined. Those at distinct control points do not hold simultaneously and can be disjoined. Thus we can summarise the information from INTERPROC as a formula in disjunctive normal form.

We denote the call to INTERPROC and the invariant parser by  $weaken(M)$ , and add assertion weakening to our method as follows

```

procedure  $mcrw(M_R, P)$ 
1. match  $mcr(M_R, P)$ 
2. case SUCCESS : return SUCCESS
3. case  $\pi : \text{let } \langle a_0, \dots, a_m \rangle = \pi$ 
4.  $dnf \leftarrow weaken(M_R \upharpoonright \pi)$ 
5. if  $(P \Rightarrow P \vee dnf) \wedge \neg(P \vee dnf \Rightarrow P)$ 
6. then  $P \leftarrow P \vee dnf$ 
7. else return FAIL
8. goto 1

```

The call to  $mcr$  may statefully refine  $M_R$ . In line 4,  $M_R \upharpoonright \pi$  is the state machine that is exactly as  $M_R$  but restricted to the actions in  $\pi$  (so it automatically inherits any control-flow refinement). Line 5 uses the CVC3 SMT solver [4] to evaluate the test. If an assertion strictly weaker than  $P$  cannot be found, the stability repair attempt declares failure.

The call to  $weaken$  operates only on the part of the environment that is causing instability. This ensures that we do not weaken  $P$  unnecessarily, and also eases the abstract interpreter's task. The call to  $mcr$  resets all pins. This slows down convergence, but also decreases the likelihood of returning an unnecessarily strong  $R$ . As noted earlier, the weaker the  $R$  and the stronger the  $P$ , the better.

An overview of the framework is given in Figure 1. Solid arrowheads indicate positive branches at decision points. Shaded boxes indicate manual work: first, the user must annotate code with actions, from which  $R$  can be automatically constructed (it is manual for now); second, the assertion to stabilise must be supplied.

## 5 Examples

We give a few simple examples to illustrate the framework and the limitations of our current implementation.

*Example 1* Let the environment be

```

1. int  $x = 1$ 
2.  $x = x + 1$ 
3.  $x = x - 1$ 
4. goto 2

```

and suppose we wish to check the stability of the stable assertion  $P \equiv x \geq 1 \wedge x \leq 2$ . In standard RG, the actions of this environment are  $x := x + 1$  and

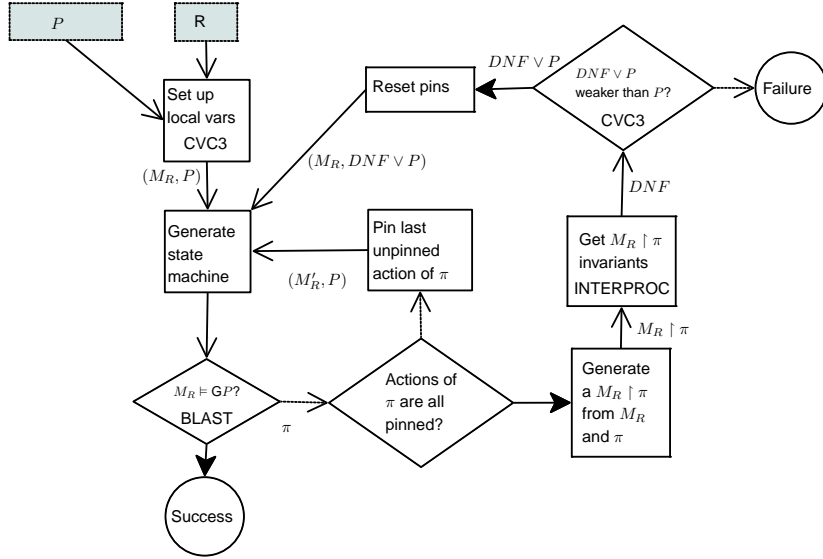


Fig. 1. Overview of Framework

$x := x - 1$  (the initialisation of  $x$  is absorbed into the initial state).  $P$  is not stable under  $R$ , which is the reflexive transitive closure of these actions, e.g.,  $R$  could consecutively execute the increment twice.

Invoking the stability analysis, the model checker fails, and the counterexample shows either  $x > 2$  because of the first action or  $x < 1$  because of the second action. The offending action is then pinned, but the second run of the model checker will fail again, because the remaining unpinned action can again occur thrice consecutively. The analysis pins this action as well, and now the check succeeds because both actions must now alternate. This example also shows how our method can prove stronger properties than are possible in standard RG, despite being automatic.

*Example 2* We now see a case which requires weakening for success. Suppose the environment is

1. `int x = 1`
2. `x = x + 1`
3. `goto 2`

and we wish to check the stability of the assertion  $P \equiv x = 1$ . This is unstable, because of the increment.

The model checker fails, and the counterexample points to  $x := x+1$ . Pinning this, the model checker fails again. Now the counterexample trace contains no unpinned actions, so the analysis concludes that there is at least one way to

destabilise the assertion even with full control-flow tracking. It calls the abstract interpretation engine for help, which in this case returns the invariants  $x - 1 = 0$  for line 1, and  $x \geq 1$  for line 3. The analysis then uses an SMT solver to confirm that  $P \vee (x - 1 = 0 \vee x \geq 1)$  is strictly weaker than  $P$ , and weakens  $P$  to  $x - 1 = 0 \vee x \geq 1$ . The model checker now succeeds even with all pins reset.

This example illustrates the use of weakening, but also exposes one of the shortcomings of the method as currently implemented. There is a tradeoff between weakening  $P$  and strengthening  $R$ , and the current approach always prioritises the latter. We are currently investigating more sophisticated algorithms that attempt to efficiently find the strongest  $P$  and weakest  $R$  for which stability is provable.

*Example 3* The above examples may have given the impression that we always end up with full control-flow tracking. But consider the environment

1. `int x = 10, y = 10`
2. `x = 10`
3. `y = 10`
4. `if x ≤ y then x = x + 1`
5. `if y ≤ x then y = y + 1`
6. `goto 4`

and let  $P \equiv x - y \leq 1 \wedge y - x \leq 1$ . The actions are

1. `x := 10`
2. `y := 10`
3. `x ≤ y ↘ x := x + 1`
4. `y ≤ x ↘ y := y + 1`

and  $P$  is not stable under the initial  $R$  (which does not track any control-flow), because the first two actions can execute any time. However, the order of the last two actions does not matter, and indeed the analysis does pin the first two only.

*Example 4* We now consider a concrete example that occurred in our own research. During the construction of a correctness proof of Simpson’s 4-slot algorithm [20], the second author devised the algorithm of Figure 2. This simulates a lock-free one-place buffer that is concurrently accessed by a single writer thread and a single reader thread. The implementation uses as shared state a 2-place array  $b$  and two integers  $l$  and  $c$ , and manages reads and writes without corruption and in sequence (but reader starvation is possible). Reads and writes are atomic at bit level only, so in particular the read and write of  $b$  are not atomic.

Safety (i.e., reader never returns a corrupt value) can be model-checked. Less obvious is *freshness*. This states that the reader must always read a value that is either the same as or occurs later in the sequence of written values, than the latest such value when the reader thread started.

write( $x : T$ )	read() returns $y : T$
<b>local</b> $t$ :int	<b>local</b> $t$ :int
$c = 1$	<b>do</b> $c = 0$
$t = 1 - l$	$t = l$
$b[t] = x$	$y = b[t]$
$l = t$	$t = c$
	<b>until</b> $t == 0$
	<b>return</b> $y$

**Fig. 2.** Lock-free one-place buffer

For freshness, we can assume without loss of generality that the values written to  $b$  are the serial numbers of the written values. Then we construct the environment from the user supplied writer actions (i.e.,  $c := 1$ ,  $b[-l] := b[l] + 1$  and  $l := -l$ ), supply the reader pre-condition  $v = b[l]$  and the post-condition  $y \geq v$ , and do a standard strongest-postconditions forward analysis, using the algorithm for stability checking. Both the environment construction and forward analysis can be automated, and we plan to do so in due course.

We expect that automatically proving this can also be done with an infinite state model checker that can handle parallel composition. It would need manual code instrumentation with observer variables (effectively simulating serial numbers), since temporal logic cannot otherwise capture this property. However, this approach is unlikely to scale as thread sizes grow, whereas our approach completely sidesteps state space explosion in the thread being verified, and will only possibly encounter state space explosion in the environment if the cause of instability is unrelated to control-flow.

## 6 Related Work

We do not know of any other work that uses model checking and abstraction for stability analysis in Hoare-style RG proofs. There is work underway at MSR Cambridge [12] that also represents  $R$  and  $G$  as state machines, but their aim is to deal with questions of liveness. Other than that we know only of the work on automatic stabilisation for a fragment of separation logic [6, 21], which uses weakening only and is restricted to syntactic checking for instability.

There has been work on modular verification using model checking, where the component being verified is distinguished from its environment (which can be abstracted and refined, e.g., [2]). Our work is both less powerful in that it requires the user to provide the environment (i.e., the rely) manually, and more powerful in that it is able to use Hoare logic, thus sidestepping the state explosion problem within the component being verified.

The work on thread-modular verification by Flanagan et al [11] is more relevant. They also use a rely-guarantee style approach, but in which executions of the rely are inserted throughout the code of the thread being verified, and

at each step, it is checked that the guarantee of the thread holds. Their work is more mature, with support for procedure calls and connections with sequential program checkers. However, their approach can be seen as a special case of our framework: the case where proof fails if it fails for the initial environment and specification.

## 7 Concluding Remarks

This paper only establishes proof-of-concept. For a start, we would like to construct the environment automatically from the annotations, so that we do not have to do our proofs one thread at a time. To handle possible environment state-space explosion in the stability detection phase, we propose the use of abstract separation logic to carve out irrelevant state. Separation logic and RG have already been combined [22]. We may also use INTERPROC supplied invariants to help the model checker with abstraction refinement.

The algorithm has other important limitations. It may not terminate. There are proof scalability issues (see §4.2). It inherits from RG the limitation to parallel composition (no forks/joins), and to proving safety properties only. Further, we completely ignore the heap.

Nonetheless, the current implementation is useful (e.g., §5), and, apart from possible non-termination, the limitations are not permanent [10, 12].

*Acknowledgement* The first author would like to thank Viktor Vafeiades for permission to reproduce excerpts from the description of RG in his Ph.D. thesis.

## References

1. M. Abadi and L. Lamport. Conjoining specifications. Technical Report 118, DEC Systems Research Center, 1993.
2. Rajeev Alur, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Automating modular verification. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR*, volume 1664 of *LNCS*, pages 82–97. Springer, 1999.
3. APRON project. <http://apron.cri.enscm.fr/>.
4. Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19<sup>th</sup> International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2007.
5. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *STTT*, 9(5-6):505–525, 2007.
6. C. Calcagno, M. J. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *LNCS*, pages 233–248. Springer, 2007.
7. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
8. E. M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification - (CAV'00)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

9. Dino Distefano, Peter O’Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.
10. Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning, 2008. Draft.
11. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.
12. Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving liveness properties of non-blocking data structures. Submitted to POPL 2008.
13. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of Computer Aided Verification (CAV ’97)*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.
14. C. A. R. Hoare. An axiomatic basis for programming. *Communications of the ACM*, 12(10):576–580, 1969.
15. Interproc static analyzer. <http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html>.
16. Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
17. Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC ’96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM Press, 1996.
18. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
19. H. Saïdi. Model checking guided abstraction and analysis. In Jens Palsberg, editor, *Proceedings of the 7th International Static Analysis Symposium*, volume 1824 of *LNCS*, pages 377–396. Springer, July 2000.
20. H.R. Simpson. Four-Slot Fully Asynchronous Communication Mechanism. *IEE Proceedings*, 137(1):17–30, 1990.
21. Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.
22. Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, volume 4037 of *LNCS*, pages 256–271, 2007.