Model Checking for Stability Analysis in 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. Rely-guarantee (RG) reasoning is useful for modular Hoare-style proofs of concurrent programs. However, RG requires that assertions be proved stable under the actions of the environment. We cast stability analysis as a model checking problem and show how this may be of use in interactive and automatic verification.

1 Introduction

Multi-core and multi-processor computing systems are now mainstream. Consequently, concurrent programs are the focus of much recent research on automatically proving safety, correctness and liveness 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 demonstration is expected to be the first step towards a fully automatic method.

Shared-memory concurrency, where multiple threads have read/write access to the same memory addresses, is commonplace. The main challenge in proving properties of such 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. Whereas locks make it easy to reason about the correctness, they may also cause loss of efficiency as threads wait to acquire locks on 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 [14].

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 it could execute in.

Rely-guarantee reasoning [11] offers a solution to this problem within the framework of Hoare-style program proofs [10], by encoding the environment into

the proof: all assertions must be shown to be unaffected under the actions of the environment. Automatically checking for and ensuring such non-interference can be problematic in many cases. In this paper, we describe preliminary progress on a possible solution.

The next section gives brief relevant background. We then describe our method, and comment on shortcomings and possible developments. We assume some familiarity with program proofs using Hoare logic [10], and with model checking [2].

2 Preliminaries

2.1 Rely-guarantee Reasoning

Rely-guarantee (RG) is a compositional verification method for shared memory concurrency introduced by Jones [11]. 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 [11], 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., threads are programs with access to some shared state, and atomic instructions occur interleaved.

Program variables will range over $\mathbb{B}$ and $\mathbb{N}$. 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 words or bit-vectors, which are harder to abstract due to fiddly problems with overflow and underflow.

RG can be seen as a compositional version of the Owicki-Gries method [15]. 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. C satisfies this specification if from a state satisfying P, and under environmental interference R (the *rely*), C causes interference at most G (the *guarantee*), and if it terminates, it does so in a state satisfying Q.

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 $P \rightsquigarrow Q$. This notation indicates that the action updates the part of shared state that satisfies P (at the moment the action executes), so that it satisfies Q.

For example, the action corresponding to the command `x := x + 1`, that increments a shared integer x, might be written as

$$x = N \rightsquigarrow x = N + 1$$

where the implicitly existentially quantified N serves to relate the state before and after the execution. Such *logical* variables are required for describing actions using single-state assertions. We shall denote them using $N, M, \ldots$ and assume they are existentially quantified with scope limited to the action.

G is the relation given by the reflexive and transitive closure of all actions of the thread being specified. The actions are given by manual annotation, as in general, automatic action discovery is non-trivial. R is calculated in an identical manner from the actions of the environment. Typically, the actions comprising R are just the G actions of all the other threads.

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. For our purpose, we do not need to pin down the level of atomicity of execution.

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 $||$ is the interleaving parallel composition operator.

$$\frac{(P_1, R \vee G_2, G_1, Q_1) \vDash C_1 \qquad (P_2, R \vee G_1, G_2, Q_2) \vDash C_2}{(P_1 \wedge P_2, R, G_1 \vee G_2, Q_1 \wedge Q_2) \vDash C_1 || C_2}$$

The second rule tells us what it means for a command to be atomic.

$$\frac{(P, id, \mathbf{true}, Q \wedge G) \vDash C \qquad P \text{ stable under } R}{(P, R, G, Q) \vDash atomic(C)}$$

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 (technically, a Kripke structure) 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 $L : S \to 2^{AP}$ 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 time-dependent properties of the program.

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

Global invariants 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 states) leading from an initial state to a state in which the invariant is not satisfied.

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) [3, 16], 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. For our purposes we will assume a simple abstraction scheme consisting of a single total *abstraction function* $\alpha : S \to A$, where A is the abstract state space (the exact structure of which depends on the α under consideration). Typically, α is not injective and need not be surjective.

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 [2, 3, 7, 16].

3 Stability Analysis as Model Checking

If the assertion permits, stability can be checked syntactically and unstable assertions can be automatically stabilised by a fix-point computation that disjunctively adds state until stability is achieved. More precisely, given an assertion

satisfying a set of states s, we compute the fix-point by

$$s_0 = s \qquad s_{n+1} = s_n \cup R(s_n)$$

until $s_{n+1} = s_n$, i.e, performing n environment transitions. If the domain of any program variable is infinite, this fix-point computation might not terminate. In such cases, automatic stabilisation techniques rely on abstract interpretation to simplify the domain.

As a simple example, consider the assertion $x = 10$ that is clearly unstable under the environment action $x = N \rightsquigarrow x = N + 1$. To stabilise, we would proceed

$$\begin{aligned} s_0 &= (x = 10) \\ s_1 &= (x = 10 \vee x = 11) \\ s_2 &= (x = 10 \vee x = 11 \vee x = 12) \\ &\vdots \end{aligned}$$

so automatic stabilisation will not terminate. To fix it, we could use the boolean abstraction $\alpha(x) \iff x \geq 10$. Under this abstraction the action above becomes the identity action 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 $x \geq 10$.

In general, if the abstraction is too weak, it may throw away so much information that the proof becomes impossible (e.g., we can trivially stabilise any assertion by replacing it with **true**). But if the abstraction is too strong, the fix-point computation may not terminate, or may run out of time or space resources.

Current techniques therefore use hand-crafted abstraction heuristics that are found to work in practice, for the underlying variable domains [5, 17].

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 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. We now present the first step towards this goal, by representing stability analysis as a model checking problem.

Rather than representing R and G as the reflexive transitive closures of their constituent actions, we consider them as state machines over the shared state. In addition, the state machine also has state variables for the program counters of the constituent threads. Tracking program counters allows us to easily encode the flow control of actions in the state machine.

The state machine for the guarantee condition G_t for a thread t, is $M_t = (S_t, S0_t, t, L_t)$ and is constructed as follows. Let V_t be the set of all shared program variables used in t as well as the t program counter $pc_t : \mathbb{N}$. Then S_t is

constructed like S in §2.2. $S0_t$ is those states of S_t in which $pc_t = 0$ and in which any other $v \in V_t$ are assigned their initial values if any. Let a_l be the (possibly empty) set of actions associated with the primitive command on line l of the thread code (this is for easy specification: in reality a single atomic statement can only have a single associated action, so the analysis simply conjoins them). Then

$$t((s, pc_t), (s', pc'_t)) = \bigvee_l \left(pc_t = l \wedge pc'_t = next(s, l) \wedge \bigwedge_{a \in a_l} a(s, s') \right)$$

where the *next* function encodes the control flow of thread t. Finally, $L_t(s) = \{p \in AP \mid p(s) = \mathbf{true}\}$. The state machine for R, $M_R = (S_R, S0_R, T_R, L_R)$ over variables V_R, is constructed analogously, though of course it is more complicated since it encompasses the actions of all the other threads.

Now suppose that we wish to stabilise an assertion P that is the pre-condition of a command at program line l in a thread t. The first step is to check whether the assertion is already stable. To do this we augment M_R with fresh[1] variables corresponding to any thread-local variables that occur in P, and also add identity actions over these variables to T_R so that their values never change. This represents our intuition that when checking the stability under the environment of an assertion in thread t, t itself is not executing.

We can now model check the augmented state machine M'_R for the global invariant P. Note that here we can use standard model checking abstraction construction techniques [7] to try and avoid non-termination. If the invariant holds, then since it is a global invariant and α is total, it holds in the concrete state space as well. Otherwise, we will obtain a counter-example giving a sequence of actions of the abstract state machine that violates the invariant. At this point, standard CEGAR techniques can be employed to check whether the counterexample has a corresponding concrete trace, in which case the stability check has failed. If not, the abstract trace is spurious (caused by too weak an abstraction), so we refine the abstraction using standard CEGAR methods and call the model checker again, until we have success or failure.

At this point we have already improved on existing stabilisation methods by not being reliant on having a syntactic check for stability. However, we still have to handle the case where the stability check fails.

In this situation, we have at hand a counterexample trace π showing a sequence of environment actions that falsified P, and also the particular abstraction function α being used by the model checker when the stability check failed. These two pieces of information can be used to weaken P and then repeat the stability check, and iterate until P is stable.

[1] So that there is no name clash with any $v \in V_R$.

For example, if we are using predicate abstraction techniques, then for our running example where we are checking stability of $P \equiv x = 10$, we may have

$$\alpha(x) \iff x = 10 \quad \text{and} \quad \pi \equiv x = 10 \rightsquigarrow_k x = 11$$

where k uniquely identifies the action responsible. This information suggests generalising P to $x \geq 10$, and then the stability check succeeds.

This is as far as we have come. The new weakened assertion must be found manually for now, using the point-of-failure α and π as guides, as in the example above. Of course, we could simply use the existing method of repeated disjunctive addition of the resultant state of the involved actions (which in this cherry-picked example fails to terminate). However, the model checking approach gives us extra information (point-of-failure π and α) which should hopefully allow us to do better. We plan to develop an automatic method that uses symbolic simulation driven by the counterexample traces, perhaps in combination with heuristics, to weaken P in a useful manner. Here, we expect to use existing model checking research on automatic abstraction construction [8].

In fact, at the moment our in-development tool (effectively a translation layer on top of the NuSMV model checker [1]) does not even perform abstraction, as all our test assertions are over finite domains. This is because while it would be simple to switch to a tool supporting automatic abstraction (e.g., BLAST [9]), we are more interested in finding out how to use π to weaken P, which is the real challenge.

3.1 Comment: Refining Stability

In standard RG, the rely is represented as the reflexive transitive closure of all actions that the environment can execute. This can also be thought of as a state machine, albeit a not very informative one in which any transition (action) can execute from any state. Thus, our representation of R and G as state machines of actions can be seen as a refinement of the standard RG representation. The latter can be thought of as the state machine consisting of all states reachable from any state via all possible interleavings of the underlying actions, regardless of whether these interleavings will ever actually occur. Our refinement proceeds by adding control flow information, thus ruling out certain interleavings.

Thus it is possible for us to prove the stability of stronger assertions than is possible in standard RG. Since the stability check is orthogonal to the RG proof system, this means that we automatically obtain a stronger proof system.

Indeed, we can parameterise the RG proof system by the level of refinement of R and G. We have experimented along these lines by adding some G actions to R, or by selectively exposing the thread-local state of the environment, both of which rule out some class of impossible action sequences. In each case we

have been able to prove properties that are stronger, and often more intuitive to specify.

There is a trade-off here, since adding more information to R and G will almost certainly make the underlying model checking problem harder, affecting scalability. Nonetheless, increasing the refinement level is attractive not only because it permits stronger properties to be proved, but also because the stabilised assertion may be syntactically smaller and thus more readable. This latter consideration is important if these methods are used as part of a larger interactive proof framework, such as a theorem prover.

4 Remarks

We do not know of any other work that uses model checking for stability analysis in Hoare-style RG proofs. There is work underway at MSR Cambridge [6] 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 automatic stabilisation work that inspired our own effort [17].

It is well known [4] that the standard RG proof rule for parallel composition can become unsound if the satisfaction relation is strengthened (e.g., to include liveness). We are safe since stability is a safety property, for which the standard RG proof system is sound.

Apart from the unfinished aspect, this approach has other shortcomings. An important one is that refining R and G quickly makes the underlying model checking problem more resource intensive. The same refinement (specifically, the need to track program counters) also prevents the results from scaling up to arbitrary numbers of threads for free, unlike in standard RG. We expect that model checking techniques like paramatric verification [13] and assume-guarantee reasoning [12] (not to be confused with rely-guarantee) may help with this. More generally, our ability to change the refinement level of R and G should also help ameliorate the situation.

We are also considering the use of separation logic in this framework, to frame out irrelevant state and thus alleviate our model checking woes. RG and separation logic have already been combined [18]. Extending that framework to our method will be another thread of future work.

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

References

1. Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveriand, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *LNCS*. Springer, March 2002.

2. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
3. 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.
4. Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. CUP, 2001.
5. 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.
6. Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving liveness properties of non-blocking data structures. Submitted to POPL 2008.
7. 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.
8. Arie Gurfinkel, Ou Wei, and Marsha Chechik. Systematic construction of abstractions for model-checking. In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 381–397. Springer, 2006.
9. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 262–274. Springer, 2003.
10. C. A. R. Hoare. An axiomatic basis for programming. *Communications of the ACM*, 12(10):576–580, 1969.
11. Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
12. K. L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *LNCS*, pages 219–234. Springer, 1999.
13. K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In Tiziana Margaria and Thomas F. Melham, editors, *Proceedings of the 11th International Conference on Correct Hardware Design and Verification Methods*, volume 2144 of *LNCS*, pages 179–195. Springer, 2001.
14. 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.
15. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
16. 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.
17. Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.
18. Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, volume 4037 of *LNCS*, pages 256–271, 2007.