

A Counterexample-Guided Refinement Tool for Open Procedural Programs^{*}

Aleksandar Dimovski¹, Dan R. Ghica², and Ranko Lazić^{1**}

¹ Department of Computer Science, Univ. of Warwick, Coventry, CV4 7AL, UK

² School of Computer Science, Univ. of Birmingham, Birmingham, B15 2TT, UK

Abstract. We present a model checking tool based on game semantics and CSP for verifying safety properties of software, such as assertion violations or array-out-of-bounds errors. The tool implements a data-abstraction refinement procedure applicable to open programs with infinite integer types. The procedure is guaranteed to terminate for unsafe inputs.

Keywords: software model checking, abstraction refinement, game semantics, CSP, FDR

1 Introduction

The traditional approach to building models of software is based on representations of *program state* and the way it changes in the course of execution. A different approach to constructing models of software is by looking at the ways in which a term can observably *interact* with its context. This modelling technique, known as *game semantics*, has been shown to provide useful algorithms for software model checking [1]. In this framework, computation is seen as a game between two players, the environment and the program, and the model of a program is given as a *strategy* for the second player. Strategies can be then given concrete representations using various automata or process theoretic formalisms, thus providing direct support for model checking.

This approach has several benefits compared with the state-based approach. First, it can be applied to open program fragments with higher-order procedures. Second, game semantics is defined recursively on syntax, therefore the model of a term is constructed from the models of its subterms, using a notion of strategy composition. Third, the generated models are fully abstract, i.e., two terms have the same models if and only if they cannot be distinguished with respect to operational tests such as abnormal termination in any program context. Finally, game models are often much smaller than state-based models because details of local-state manipulation are hidden during strategy composition.

The traditional, state-based, approach to software model checking has been applied successfully to verifying realistic industrial software. At the heart of

^{*} This research was supported by the EPSRC (GR/S52759/01).

^{**} Supported by a grant from the Intel Corporation. Also affiliated to the Mathematical Institute, Serbian Academy of Sciences and Arts, Belgrade.

many successful tools such as SLAM [2] and BLAST [6] are algorithms based on abstract-check-refine loops [3]. Recently, it has been shown how counterexample-guided refinement ideas can be adapted to the setting of game-semantic models [5]. However, implementing the procedure in [5] is non-trivial because the semi-algorithm, as described, is highly inefficient.

In this paper, we describe GAMECHECKER, a tool which implements efficiently an abstraction refinement procedure for checking whether a program fragment is unsafe, i.e. it *may* execute the designated unsafe command `abort`. Using this special command, we can easily perform various code-level checks for errors such as buffer overruns or assertion violations. Semantically, this corresponds to reachability, in the model, of the designated unsafe move *abort*.

GAMECHECKER adds abstraction annotations to the source code, to approximate infinite integer data types by partitionings. Any partitioning consists of finitely many partitions, which are called *abstracted integers*. Any abstracted integer is thus a set of integers. Terms which use abstracted rather than actual integers are called *abstracted terms*. Operationally, abstracted terms behave just like their concrete counterparts, but first they nondeterministically instantiate any abstracted integer argument a by choosing nondeterministically some concrete integer $n \in a$, then abstract the concrete integer result n' to the partition $a' \ni n'$ to which it belongs. As shown in [5], this is a conservative approximation. By quotienting over abstracted integers, the models become finite and can be model-checked.

Within GAMECHECKER, an abstracted term is compiled into a process in the CSP process algebra (e.g. [7]), whose finite-traces set represents the quotiented game-semantic model of the term. The resulting process is then verified for safety using the FDR refinement checker, which is based on explicit state enumeration.³ If no counterexample is found by FDR, the procedure terminates with answer SAFE. Otherwise, the counterexamples are analysed and classified as either genuine or (potentially) spurious. If genuine counterexamples exist the program is deemed UNSAFE, otherwise the spurious counterexamples are used to refine the abstractions, by splitting some of their partitions. The procedure is then repeated on the refined term.

The abstraction refinement procedure is a semi-algorithm: it terminates and reports a genuine error trace for unsafe terms, but it may diverge for safe terms.

GAMECHECKER is available from:

<http://www.dcs.warwick.ac.uk/~aleks/gamechecker.htm>.

2 The programming language and its game semantics

The input is any program fragment of an expressive programming language combining imperative features, locally-scoped variables and (call-by-name) procedures. The actual language on which GAMECHECKER works also incorporates abstraction annotations, which are managed automatically by the tool. The data

³ FDR is a commercial product of Formal Systems (Europe) Ltd. It is available free of charge for academic use. See <http://www.fs1.com>.

types are booleans and abstracted integers ($\tau ::= \text{bool} \mid \text{int}_\pi$). The phrase types are types of expressions, variables and commands ($\sigma ::= \text{exp } \tau \mid \text{var } \tau \mid \text{com}$), and 1st-order functions types ($\theta ::= \sigma \mid \sigma \rightarrow \theta$).

The abstractions π range over computable finite partitionings of the integers \mathbb{Z} . The tool currently uses the following abstractions:

$$[] = \{\mathbb{Z}\} \quad [n, m] = \{<n, \{n\}, \{n+1\}, \dots, \{0\}, \dots, \{m-1\}, \{m\}, >m\}$$

where $<n = \{n' \mid n' < n\}$ and $>n = \{n' \mid n' > n\}$. Instead of $\{n\}$, we may write just n . Abstractions are refined by *splitting* abstract values: $[]$ is refined to $[0, 0]$ by splitting \mathbb{Z} ; $[n, m]$ to $[n-1, m]$ by splitting $<n$, or to $[n, m+1]$ by splitting $>m$.

We write $\Gamma \vdash M : \theta$ to indicate that term M with free identifiers in Γ has type θ . (The typing rules can be found in [5].) A context (i.e. term-with-hole) is *safe* if it does not contain the abort command. A term is *unsafe* if there exists a closed program formed from a safe context and the term, which may execute abort. Otherwise, we say that a term is *safe*.

GAMECHECKER includes a compiler from any abstracted term $\Gamma \vdash M : \theta$ to a CSP process $\llbracket \Gamma \vdash M : \theta \rrbracket$ whose set of finite traces $\mathbf{traces}[\llbracket \Gamma \vdash M : \theta \rrbracket]$ is the set of all plays of the game strategy for the term. Those processes are defined compositionally, by induction on the structure of terms (see [4]).

The abstraction refinement procedure described in [5] requires models consisting of *fully revealed plays*, i.e., models in which semantic composition of strategies does not involve *hiding* of the moves involved in composition. The fully revealed plays allow us to discern between genuine and spurious counterexamples by identifying the precise subterms that produce abstracted moves. However, fully revealed models are much larger and therefore impractical. In GAMECHECKER, this is overcome as follows: first we use special marker moves to identify points in plays at which abstraction gives rise to nondeterminism, then we use a special debugging feature of FDR that lets us reveal only those plays which are counterexamples rather than full models.

Nondeterminism due to abstraction happens when an arithmetic/logic operation produces more than one result. In such an instance, the operation necessarily has at least one abstracted integer operand which is not a singleton, i.e. which abstracts more than one integer. The game strategy for the operation then performs a special marker move $nd.a$, where a is such an operand. Those moves are propagated through strategy compositions, so for any term $\Gamma \vdash M : \theta$, they appear in $\mathbf{traces}[\llbracket \Gamma \vdash M : \theta \rrbracket]$ at the points where nondeterminism due to abstraction occurs.

Example 1. Consider $\llbracket x : \text{var int}_{[0,4]} \vdash x := x + 1 : \text{com} \rrbracket$. If the abstract value <0 is read from x , $x + 1$ can evaluate to both 0 and <0 . The following trace corresponds to choosing the result 0: $run\ read_x\ <0_x\ nd.<0\ write(0)_x\ ok_x\ ok$. The move $nd.<0$ records the non-singleton abstracted integer operand <0 .

FDR offers a number of state-space reduction algorithms which preserve finite-trace sets, and which are thus compositional. The processes representing the game strategies are particularly amenable to such reductions, because

moves which are hidden through composition of strategies become internal (τ) process transitions. The compiler within `GAMECHECKER` inserts calls to FDR’s state-space reduction algorithms within the process scripts it outputs.

It was established in [5] that the game semantic models are fully abstract for the language of concrete and abstracted terms. This result ensures that, for any term $\Gamma \vdash M : \theta$, model-checking the process $\llbracket \Gamma \vdash M : \theta \rrbracket$ for safety (i.e. for unreachability of the *abort* event) is equivalent to checking whether $\Gamma \vdash M : \theta$ is safe.

3 Abstraction refinement procedure

`GAMECHECKER` checks safety of a given term $\Gamma \vdash M : \theta$ (with infinite integer data types) by performing a sequence of iterations. The initial abstracted term $\Gamma_0 \vdash M_0 : \theta_0$ uses the coarsest abstraction \mathbb{Z} for any free identifier or local variable, and the abstraction $[0, n]$ or $[n, 0]$ for constants n . Other abstractions (such as those for integer expression subterms) are determined from the former by inference.

Each iteration consists of model checking (by calling the FDR tool), slicing, and refining abstractions. Only abstractions which occur in types of free identifiers or local variables are explicitly refined, and others are obtained by inference. That yields a refined abstracted term $\Gamma_{i+1} \vdash M_{i+1} : \theta_{i+1}$, which is passed to the next iteration.

The following are the steps of any iteration. If t is a trace which contains at least one special move marking a nondeterminism, let $\mathbf{pick}(t) = a$, where $nd.a$ is the first such move.⁴ For ordering non-singleton abstracted integers, we use a bijection r to the natural numbers: $r(\mathbb{Z}) = 0$, $r(<n) = 2|n| + 2$, and $r(>n) = 2n + 1$. This has the property that $r(a) < r(a')$ whenever $a' \subseteq a$.

- 1 If $\llbracket \Gamma_i \vdash M_i : \theta_i \rrbracket \setminus \{nd\}$ is unsafe, terminate with answer UNSAFE.
- 2 If $\llbracket \Gamma_i \vdash M_i : \theta_i \rrbracket$ is safe, terminate with answer SAFE.
- 3 Among the counterexamples (i.e. traces of $\llbracket \Gamma_i \vdash M_i : \theta_i \rrbracket$ which end with *abort*), select t such that $r(\mathbf{pick}(t))$ is minimal.
- 4 Apply the FDR trace-reveal feature to t , obtaining a fully revealed trace s .
- 5 Call a slicing procedure to determine a set S of all occurrences of non-singleton abstracted integers which were involved in causing the first $nd.a$ move in s .
- 6 For any data type int_π of a free identifier or a local variable which corresponds to an occurrence of an abstracted integer b in S , refine π by splitting b .

Steps 2 and 3 are implemented as follows. The process $\llbracket \Gamma_i \vdash M_i : \theta_i \rrbracket$ is composed in parallel with an auxiliary process *Rank_of_pick* which, once the first move of the form $nd.a$ has occurred, keeps in its state the value $r(a)$. FDR is called to model check that parallel composition, and for any reachable state which has

⁴ This definition of $\mathbf{pick}(t)$ is currently implemented, but other definitions are possible. The crucial property is that, if t is used to refine abstractions, then one of the refinements will split $\mathbf{pick}(t)$.

an *abort* transition, to return a trace which reaches it. By step 1, any such trace must contain an *nd.a* move. The parallel composition with *Rank_of_pick* ensures that, for any possible value of $r(\mathbf{pick}(t))$ with t a counterexample, at least one such counterexample is returned by FDR.

Theorem 1. *If the abstraction refinement procedure terminates, its answer is correct. Moreover, it terminates for any unsafe term.*

Proof. UNSAFE answers are correct because any trace which contains no special moves marking nondeterminism corresponds to a concrete trace. Correctness of SAFE answers is a consequence of the conservativity of abstraction. (See [5].)

Suppose $\Gamma \vdash M : \theta$ is unsafe. Let u be a fully revealed play of the game strategy for $\Gamma \vdash M : \theta$ which ends with *abort*, and let m be an integer in u with maximum absolute value. For any non-singleton abstracted integer a , we define $d(a) = 2|m| + 1 - r(a)$. Then $d(a) > 0$ whenever $|n| \leq |m|$ for some $n \in a$.

For any iteration i , let D_i be the sum of all positive $d(a)$ as a ranges over the non-singleton equivalence classes of all abstractions in $\Gamma_i \vdash M_i : \theta_i$. Steps 3–6 ensure that $D_0 > D_1 > \dots$, so the procedure must terminate. \square

4 Conclusion

This paper presents the first software model checker based on game semantics and counterexample-guided abstraction refinement. By combining the two theories, it can handle arbitrary open program fragments with infinite integer data types. The tool is a prototype implementation, which has been tested on a variety of academic examples.

Possibilities for future work include extensions to programs with concurrency, recursion and 2nd-order procedures, as well as to abstractions by arbitrary predicates. The goal is a tool which uses game semantics to achieve compositional verification of practical programs.

References

1. S. Abramsky, D.R. Ghica, A. Murawski and C.-H.L. Ong. Applying Game Semantics to Compositional Software Modeling and Verification. In Proceedings of TACAS, LNCS **2988**, (2004), 421–435.
2. T. Ball and S.K. Rajamani. The SLAM Toolkit. In Proceedings of CAV, LNCS **2102**, (2001), 260–264.
3. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. J. ACM **50**(5), (2003), 752–794.
4. A. Dimovski and R. Lazić. CSP Representation of Game Semantics for Second-Order Idealized Algol. In Proceedings of ICFEM, LNCS **3308**, (2004), 146–161.
5. A. Dimovski, D.R. Ghica and R. Lazić. Data-Abstraction Refinement: A Game Semantic Approach. In Proceedings of SAS, LNCS **3672**, (2005), 102–117.
6. T.A. Henzinger, R. Jhala, R. Majumdar and G. Sutre. Software Verification with Blast. In Proceedings of SPIN, LNCS **2648**, (2003), 235–239.
7. A.W. Roscoe. Theory and Practice of Concurrency. Prentice Hall, 1998.

During the demonstration, we shall first show how GAMECHECKER can be used in general, and then go through several examples, including the ones discussed below. The examples will be used to show the main distinctive features of software model checking based on game semantics from a user's point of view: how models are generated and minimised compositionally, and how counterexamples consist of events which represent interactions between the term and its environment. To assist the presentation, some of the models generated will be displayed graphically.

A Using the tool

GAMECHECKER has been developed in Java.

The front end can be seen in Figure 1. The inputs are a term and a property, given as an error move whose reachability will be checked. The default error move is *abort*. The result pane shows the iteration steps, including all reported nondeterministic counterexamples and applied refinements. If the procedure terminates, it is reported whether the term is safe or unsafe. In the latter case, a genuine counterexample is returned.

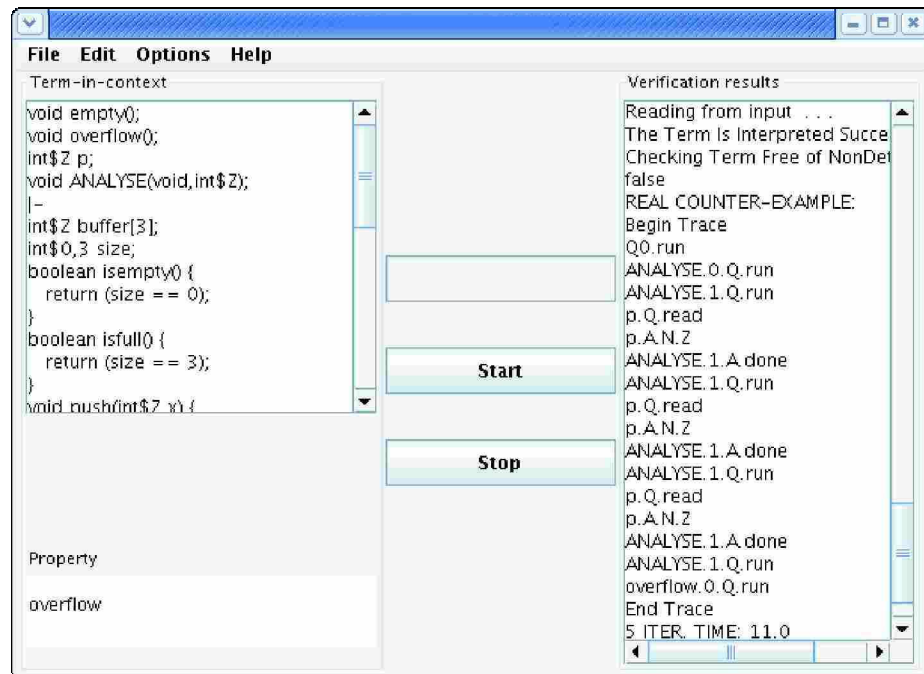


Fig. 1. A screen-shot of the tool

By default, GAMECHECKER executes a simpler abstraction refinement procedure than the one presented in Section 3, where any shortest counterexample is selected in step 3. This variant is more efficient per iteration, but it might not terminate for unsafe terms. The semi-terminating procedure is run by checking the Semi-Termination box in the Options menu.

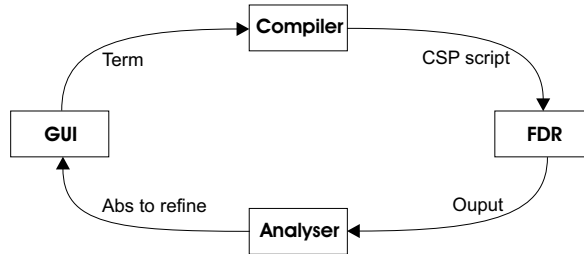


Fig. 2. The tool architecture

B Examples

The following are three progressively more involved examples. Further examples can be found at the GAMECHECKER website:

<http://www.dcs.warwick.ac.uk/~aleks/gamechecker.htm>.

B.1 A warm-up example

Consider the term

$$f(\text{com}, \text{com}) : \text{com} \vdash \text{new int } x := 0 \text{ in } f(x := x + 1, \text{if } x > 1 \text{ then abort})$$

which uses a local variable x , and a *non-local* function f . We want to check whether this term is safe from terminating abnormally. The procedure-call mechanism is by-name, so every call to the first argument increments x , and any call to the second uses the new value of x . So there are functions f for which the program is not safe, and this is how GAMECHECKER will discover the bug.

The initial abstracted term is

$$\text{new int}[\] x := 0 \text{ in } f(x := x + 1, \text{if } x > 1 \text{ then abort})$$

A nondeterministic counterexample is identified by FDR, corresponding to a function that evaluates its second argument:

$$\text{run run}_f \text{ run}_{f,2} \text{ nd.}\mathbb{Z} \text{ abort}$$

Next, by processing the detailed output from FDR, we reconstruct a corresponding fully revealed play:

$$\begin{aligned} & \text{run run}_f \text{run}_{f,2} \text{run}_{\langle 2 \rangle} q_{\langle 2,1 \rangle} q_{\langle 2,1,1 \rangle} \text{read}_{\langle 2,1,1,1 \rangle} \text{read}_x \mathbb{Z}_x \\ & \mathbb{Z}_{\langle 2,1,1,1 \rangle} \mathbb{Z}_{\langle 2,1,1 \rangle} q_{\langle 2,1,2 \rangle} 1_{\langle 2,1,2 \rangle} \text{tt}_{\langle 2,1 \rangle} \text{nd}.\mathbb{Z} \end{aligned}$$

The slicing procedure starts by examining the subterm with coordinates $\langle 2, 1 \rangle$, whose answer move precedes nd . The coordinates specify the path in the syntax tree of the term to reach the subterm, in this case the boolean expression of if . The nd move marks that this term has been evaluated nondeterministically. Since this subterm represents a logic operation, the slicing procedure is called recursively to examine its operands, i.e. terms $\langle 2, 1, 1 \rangle$ and $\langle 2, 1, 2 \rangle$. The answer move of the $\langle 2, 1, 1 \rangle$ term is the abstract value \mathbb{Z} , so the examination proceeds for its subterm $\langle 2, 1, 1, 1 \rangle$. Here, it will be detected that the $\langle 2, 1, 1, 1 \rangle$ term is a de-referencing of x and that the abstract value \mathbb{Z} is read from x . Thus, the slicing procedure will indicate that the abstraction of x needs to be refined.

The second iteration uses the refined term:

$$\text{new int}[0, 0] x := 0 \text{ in } f(x := x + 1, \text{if } x > 1 \text{ then abort})$$

Another nondeterministic counterexample is found, which represents a function evaluating its first and then its second argument:

$$\text{run run}_f \text{run}_{f,1} \text{ok}_{f,1} \text{run}_{f,2} \text{nd}.(>0) \text{abort}$$

The corresponding fully revealed play is:

$$\begin{aligned} & \text{run run}_f \text{run}_{f,1} \text{run}_{\langle 1 \rangle} q_{\langle 1,2 \rangle} q_{\langle 1,2,1 \rangle} \text{read}_{\langle 1,2,1,1 \rangle} \text{read}_x 0_x 0_{\langle 1,2,1,1 \rangle} 0_{\langle 1,2,1 \rangle} \\ & q_{\langle 1,2,2 \rangle} 1_{\langle 1,2,2 \rangle} 1_{\langle 1,2 \rangle} \text{write}.1_{\langle 1,1 \rangle} \text{write}(>0)_x \text{ok}_x \text{ok}_{\langle 1,1 \rangle} \\ & \text{ok}_{\langle 1 \rangle} \text{ok}_{f,1} \text{run}_{f,2} \text{run}_{\langle 2 \rangle} q_{\langle 2,1 \rangle} q_{\langle 2,1,1 \rangle} \text{read}_{\langle 2,1,1,1 \rangle} \text{read}_x, (>0)_x \\ & (>0)_{\langle 2,1,1,1 \rangle} (>0)_{\langle 2,1,1 \rangle} q_{\langle 2,1,2 \rangle} 1_{\langle 2,1,2 \rangle} \text{tt}_{\langle 2,1 \rangle} \text{nd}.(>0) \end{aligned}$$

Similarly as in the previous iteration, the analyser starts exploring the non-deterministic term $\langle 2, 1 \rangle$. By searching for non-singleton abstract integers recursively through its subterms, it will detect that the abstract value >0 read from x has caused the nondeterminism. So, further refinement of x will be recommended.

The third iteration term is:

$$\text{new int}[0, 1] x := 0 \text{ in } f(x := x + 1, \text{if } x > 1 \text{ then abort})$$

Now, a genuine unsafe trace is detected: f increments x twice, then evaluates its second argument:

$$\text{run run}_f \text{run}_{f,1} \text{ok}_{f,1} \text{run}_{f,1} \text{ok}_{f,1} \text{run}_{f,2} \text{abort}$$

The model generated for the third iteration term is shown in Figure 3, where dashed edges indicate moves of the Opponent (i.e. the environment), and solid edges moves of the Player (i.e. the term). Observe that, in this case, the model contains no nd moves.

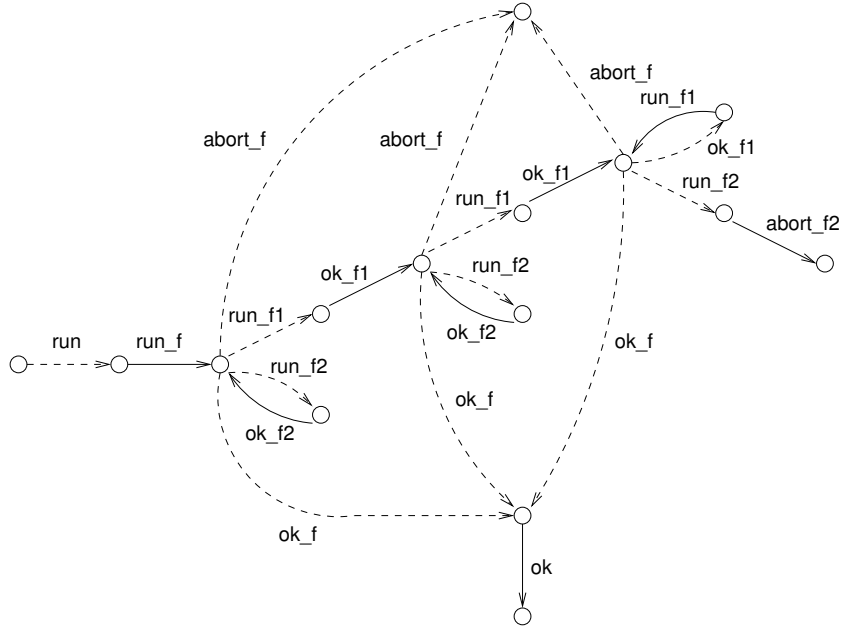


Fig. 3. A strategy labelled transition system

B.2 A semi-termination example

The term

$$\begin{aligned}
 & e : \text{int}, f(\text{com}, \text{com}) : \text{com} \vdash \\
 & \text{new int } x := e \text{ in new int } y := 0 \text{ in} \\
 & \text{if } (x == x + 1) \text{ then abort else } f(y := y + 1, \text{if } (y > 1) \text{ then abort})
 \end{aligned}$$

is an example of an unsafe term for which the simpler abstraction refinement procedure, where any shortest counterexample is selected in step **3**, does not terminate: it keeps refining the abstraction of x . If the tool is instructed to perform the semi-terminating procedure presented in Section 3, then after a few iterations of refining the abstraction of x , the abstraction of y will be refined. For this particular example, a genuine counterexample is reported after refining the abstractions of e and x to $[-2, 1]$, and the one for y to $[0, 1]$.

B.3 A stack example

Consider the following implementation of a stack of maximum size n (a meta variable). After implementing the stack by a sequence of local declarations, we export the functions $push(x)$ and pop by calling *ANALYSE* with arguments $push(p)$ and pop . In effect, the model contains all interleavings of calls to $push(p)$ and pop , corresponding to all possible behaviours of the non-local expression p and non-local function *ANALYSE*.

```

p : exp int, ANALYSE(com, exp int) : com ⊢
new int buffer[n] := 0 in new int top := 0 in
let com push(int x) {
  if (top == n) then abort else {buffer[top] := x; top := top + 1} in
let exp int pop {
  if (top == 0) then abort else {top := top - 1; return buffer[top + 1]} in
ANALYSE(push(p), pop)

```

We can check a range of properties of the stack implementation. By varying the term slightly, we can check separately for ‘empty’ (reads from empty stacks) and ‘overflow’ (writes to full stacks) errors, both of which are present for any n . We can also check that array-out-of-bounds errors are not present in the term: arrays are syntactic sugar, in which `abort` is executed if an array is referenced out of its bounds.

Table 1 contains the experimental results for checking the three properties on the stack implementation. We ran `GAMECHECKER` on a Research Machine AMD Athlon 64(tm) Processor 3500⁺ with 2GB RAM. We list the number of iterations and the execution times in minutes for different values of n . Abstraction and abstraction-refinement are crucial in generating models that are small enough to be analysed, by ensuring that the the contents of the array can be disregarded (the initial abstraction `[]` is not refined), and that the local variable tracking the top of the stack (top) is automatically adjusted to a small but safe domain (in practice, $[0, n]$).

Table 1. Experimental results for checking a stack implementation

n	empty		overflow		oub	
	Iterations	Time (min)	Iterations	Time (min)	Iterations	Time (min)
5	2	0.3	7	1.2	7	1.5
10	2	0.9	12	5.6	12	8
15	2	2	17	18	17	23
20	2	4	22	47	22	59
30	2	11	32	190	32	230
50	2	60	52	1570	52	1831
100	2	630	failed		failed	