Instrumenting C programs with Nested Word Monitors *

Swarat Chaudhuri and Rajeev Alur

University of Pennsylvania

1 Introduction

In classical automata-theoretic model checking [6], a system model generates a language L of words modeling system executions, and verification involves checking if $L \cap L' = \emptyset$, L' being the language of words deemed "unsafe" by the specification. This view is also used in program analyzers like BLAST [5] and SLAM [2], where a specification is a word automaton (or monitor) with finite-state control-flow that accepts all "unsafe" program executions. Typical analysis constructs the "product" of a program and a monitor, in effect *instrumenting* the program with extra instructions, so that the input program fails its specification iff the product program fails an assertion. The latter is then checked for possible assertion failures. Monitors also find use in testing and runtime verification, where we try finding assertion violations in the product program at runtime.

One shortcoming of these notations is expressiveness. As finite automata cannot argue about the nested structure of procedure calls and returns in programs, these languages cannot state pre/post-conditions arising in specification languages like JML [4]: "if a file is open right before a call, then it is open when the procedure returns." Nor can they reason about procedural contexts and express properties like: "if a file is opened, it must be closed before the current context ends." Another issue is that these notations cannot reason modularly about programs. If programs are structured, why not specifications as well?

These problems can be overcome if a program execution is modeled as a *nested word* [1] rather than as a word. A nested word is obtained by adding, to a word modeling an execution, a set of nested *jump-edges* that connect call sites to their matching returns. A *nested word automaton* processing a nested word reads the symbols in the underlying word just like a word automaton. However, it also takes jump-edges into account: while transitioning to a return position (a point in the word with an incoming jump-edge), a nested word automaton can consult its state at the source of the jump-edge, i.e., the call matching the return. Intuitively, the monitor tracks the program's *global control flow* by following the underlying word, and its *local control flow* by following the jump-edges.

This paper presents a specification language—called PAL—based on nested word automata, and a tool to instrument C code using it. This language extends the BLAST specification language [3], and while its richer foundations lets it state context-sensitive properties, it has syntax close to BLAST's and allows

* This research was partially supported by ARO URI award DAAD19-01-1-0473 and NSF award CCR-0306382.

easy instrumentation. Monitors in PAL are independent of the programs they are used to instrument, and work irrespective of whether recursion is present. While they are theoretically only as expressive as monitors in BLAST in the absence of recursion, they are more modular, succinct and comprehensible even in this case. We believe, therefore, that these monitors present an example of *structured specifications*, suitable for structured programs. Finally, while our monitors extend the specification format of a model checker, their use is not limited to static checking. Once a program has been instrumented with a monitor, it can be used for testing or run-time verification as well as static analysis or model checking.

2 Language description

```
global int infoo = 0;
global int open = 0;
global FILE * stream;

event {  /* event 1 */
 pattern {
   $1=fopen($2,$?);}
 guard { strcmp($2,''dat'')
   ||(open==0 && infoo==1)}
 action {
   if (!strcmp($2,''dat'')){
     open = 1;
     stream = $1;}}
}

event {  /* event 2 */
  pattern { fclose ($1); }
  action { if ($1 == stream)
             open = 0; }}

event {  /* event 3 */
  pattern { $? = foo ($?); }
  local int stored;
  before {
    action { stored = infoo;
             infoo = 1; }}
  after {
    guard { open == 0 }
    action {
      infoo = stored; }}
}
```

Fig. 1. A PAL specification

We present the PAL language using an example. Consider a C program that opens and closes files via calls to `fopen` and `fclose`, and the following requirement: "a secret file `dat` is not opened outside the scope of a file-handling routine `foo`. If `foo`, or a procedure called transitively from it, opens a stream for `dat`, then: (1) no new stream for `dat` is opened without closing the current stream, and (2) any open stream for this file must be closed by the time the top-level call to `foo` returns." Such a discipline follows programmer intuition and prevents security flaws where the main context, unaware that `foo` has left open a sensitive stream, invokes an untrusted program that can now do I/O on the "leaked" stream (for a real instance of such a bug, see Sec. 3).

A PAL monitor for this requirement is shown in Fig. 1. The states of the monitor are encoded by a set of *monitor variables*, and its transitions by a set of `event{...}` blocks. Some monitor variables are *global* and are declared using the keyword `global` — intuitively, global monitor variables may be tested or updated by any event. In addition, each event includes an optional set of *local monitor variables*, declared using the keyword `local`, whose scope is restricted to the current event.

Events are fired by matching *patterns* on statements in the analyzed program. A pattern, specified in a `pattern{...}` block, is an assignment or procedure call with possible "pattern variables" ($?, $1, $2,

etc.). During matching, the variables $1, $2, etc. match arbitrary C expressions and the variable $? serves as a wildcard— e.g., the pattern in event 1 matches all calls to `fopen`. For each statement matching the pattern [1] specified in the i-th event, the monitor sets up a precondition and a postcondition using the code in the blocks `before{...}` and `after{...}` in this event. The precondition (similarly, postcondition) checks whether an optional guard—a C expression over monitor and pattern variables, inside a `guard{...}` block—is satisfied by the monitor state right before (after) this statement. If the guard is not satisfied, an assertion violation is reported. Otherwise, the state of the monitor is updated by executing the C code contained within an optional `action{...}` block. This code is allowed to read pattern variables, and read or update monitor variables. For succinctness, we allow guards and actions to be defined outside `before` or `after` blocks (event 1 or 2)—in this case they are assumed to define preconditions.

```
int infoo = 0;
int open = 0;
FILE * stream;

bar() {
  int stored;
  ...
  stored = infoo;
  infoo = 1;
  x = foo(y);
  if (open == 0)
     infoo = stored;
  else ERROR;
  ...
}
```

Fig. 2. Instrumenting using event 3

During instrumentation, code blocks implementing an event's precondition and postcondition are respectively injected before and after statements matching its pattern. Consider a call `x = foo(y)` in a procedure `bar` in a program; on instrumentation using the monitor in Fig. 1, this line is replaced by the chunk of code in Fig. 2. Declarations of the monitor variables are added as well; `stored` is declared locally in `bar`, and `infoo`, `open`, and `stream` are declared globally.

Note that this syntax closely resembles that of the Blast query language. Blast, too, allows injection of code before or after a program statement using the keywords `before` and `after`. This similarity is a design feature, as our goal was to extend Blast minimally to obtain a specification language for context-sensitive requirements. The key new features in Pal are local variables and the ability to declare `before` *and* `after` blocks in the same event. This modification makes a major semantic difference: the control-flow of a monitor is now given by a nested word automaton, rather than a word automaton. Consider our example monitor and an execution of the input program containing a call to `foo`. In the nested word capturing this execution, there is a jump-edge from the call to `foo` to its matching return. Now, as the monitor reads this execution, it can save its state right before control enters `foo` using its local variables, and retrieve this state at the matching return. Thus, it has the power of a nested word automaton that reads the corresponding nested word, consulting its state at the source of an incoming jump-edge while transitioning to a return position. On the other hand, our monitor can use its global variables to pass states *into* invoked procedures such as `foo`, just like a Blast monitor. More abstractly, this amounts to state updates as it reads the underlying word structure.

[1] Monitors are deterministic—i.e., if more than one pattern is matched at any point, we break the tie by picking the one in the event defined first.

We end this section with some hints to check that the monitor in Fig. 1 specifies our original requirement. The variables `infoo` and `open` track whether `foo` is in the stack and whether `dat` is open, and `stream` stores a possible open stream for `dat`. The variable `stored` is used to infer whether control is back to the top-level context calling `foo`. The rest is easily verified.

3 Implementation and case studies

We have implemented PAL on top of the current implementation of BLAST. The specification and analysis modules in BLAST are orthogonal: the former generates C code instrumented with a monitor, while the latter checks the generated code for assertion failures. We extend BLAST's specification module to permit PAL monitors, and analyze the generated code statically as well as dynamically. The source code of our implementation, along with the examples that we now discuss, is available at `http://www.cis.upenn.edu/~swarat/tools/pal.tar.gz`.

File descriptor leak in `fcron` A monitor as in Sec. 2 could be used to prevent a reported bug (`http://nvd.nist.gov/nvd.cfm?cvename=CVE-2004-1033`) in Version 2.9.4 of `fcron`, a periodic command scheduler for Linux. Here, the main function of a binary (`fcrontab`) calls a routine `parseopt`, which calls a routine `is_allowed` to check if a user is "allowed", which calls a procedure that opens, but forgets to close, a stream for a secret file `fcron.allow`. After control returns to the main context, the program starts a process with a name derived from an environment variable. However, an attacker can change the value of this variable to start a malicious program that reads `fcron.allow` via the open file stream.

This error may be prevented by a policy that allows `parseopt` to open `fcron.allow`, but not to leak its descriptor. Also, we could require that this secret file is not opened outside the scope of `parseopt`. This policy makes intuitive sense: as `parseopt` is a routine verifying a username, it is reasonable that it, or procedures it calls transitively, opens the file of allowed users. However, by the principle of least privilege, this file should only be opened when necessary, i.e., when `parseopt` is on the stack. A monitor expressing these requirements looks very similar to the one in Fig. 1. On instrumenting `fcron` with this monitor, we find a policy violation within a few random tests. However, abstraction-based model checking using BLAST is not suitable for this example, as BLAST cannot currently perform good analysis of library functions like `strcmp`.

Stack-sensitive security properties Consider the security property: "A program must not execute a sensitive operation `write` at any point when an untrusted routine `foo` is on the stack." In the Java and C# languages, such policies are automatically enforced by the run-time environment, using the mechanism of *stack inspection*. In C, they may be enforced dynamically using a monitor—however, traditional monitors cannot express such properties of the stack, so that a nested word monitor is needed. Of course, such monitors could also be used in static analysis or software model checking.

Fig. 3 shows a monitor for this property. The global variable `infoo` tracks if `foo` is in the stack, and a guard prevents writes within the scope of `foo`.

We note that PAL may also be used to state some requirements of this nature that *cannot* be enforced via stack inspection. Consider the property: "If an untrusted procedure has ever been on the stack, a certain sensitive operation must not be executed." The rationale is that an untrusted routine may cause a side-effect that proves to be dangerous at a future point, so that if we call one, we must strengthen the security policy. However, since the culpable routine may no longer be on the stack when a violation occurs, stack inspection does not help in this case. On the other hand, it is easy to state such properties in PAL; code for a sample monitor is available on our webpage.

```
global int infoo = 0;

event {
  pattern { write(); }
  guard { infoo == 0 }
}

event {
  pattern {$? = foo($?);}
  local int stored;
  before {
    action { stored = infoo;
             infoo = 1; }}
  after {
    action {infoo = stored;}
  }
}
```

Fig. 3. Stack-sensitive security

Logging policies PAL also finds use in stating *logging policies* enforced in large development efforts such as Windows. Consider the property: "Whenever a procedure returns an error value, the error must be logged via a routine `log` before control leaves the current procedural context." Now, different development groups may call `log` via different wrapper functions; however, the logging policy is fixed across groups and thus independent of the wrappers. In order to track if control has returned from a wrapper to the original context, we need a PAL monitor.

While we do not have access to industrial code bases where such policies are most natural, we have applied a PAL monitor for this property on a couple of hand-coded examples. These may be downloaded from our webpage.

Acknowledgement: We thank Zhe Yang for valuable suggestions.

References

1. R. Alur and P. Madhusudan. Adding nested structure to words. In *Developments in Language Theory*, pages 1–13, 2006.
2. T. Ball and S. Rajamani. The SLAM toolkit. In *Computer Aided Verification, 13th International Conference*, 2001.
3. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *SAS*, pages 2–18, 2004.
4. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, R. Leino, and E. Poll. An overview of JML tools and applications. In *Workshop on Formal Methods for Industrial Critical Systems*, pages 75–89, 2003.
5. T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV*, pages 526–538, 2002.
6. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.