SLIC: a Specification Language for Interface Checking (of C)

Thomas Ball and Sriram K. Rajamani
Software Productivity Tools
Microsoft Research
http://www.research.microsoft.com/slam/

January 10, 2002

Technical Report
MSR-TR-2001-21

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

SLIC: a Specification Language for Interface Checking (of C)

Thomas Ball and Sriram K. Rajamani
Software Productivity Tools
Microsoft Research
http://www.research.microsoft.com/slam/

January 10, 2002

1 Introduction

Modern software systems are built by a multitude of programmers using *application program interfaces (APIs)*. When a software system is built using APIs, there are several classes of problems that can hamper its dependability: a client P of an API may use it improperly; an implementation L may not properly implement the API. There are many requirements on both the client and implementer of an API that are typically stated only in the documentation for the API. Currently, only a small portion of these requirements —namely, the number of arguments of a function, and the types of each argument and return value— are stated in the header file for the API and checked for agreement at call sites by the compiler. We wish to express temporal safety requirements [15] on the API, such as rules about ordering of function calls with associated constraints on the data values visible at the API boundary, and automatically check (statically or dynamically) if these requirements are satisfied by the client and the implementer of the API.

Let L be a library that implements a certain API (as specified by the type, variable and function declarations in a C "dot-H" file) and let P be a client of L. Client P makes calls to procedures defined in the API that are implemented by L. Additionally, the library L may make calls back to P via function pointers passed from P to L. We denote the (sequential) composition of P and L by $P \,||\, L$.

In this paper, we present SLIC, a low-level specification language designed to specify the temporal safety properties of APIs implemented in the C programming language. A SLIC specification S defines a state machine[1] that *monitors* the execution of behavior of the program $P \,||\, L$ at the API's procedural interface. The atomic "propositions" of a SLIC specification are boolean functions over the *interface state* at the API boundary between P and L. An interface state is a triple $(A, \{call, return\}, \Omega)$, where A is a procedure (named directly or indirectly in the API), the second component indicates that control is being passed to A by a call or that control is returning from A to its caller, and Ω is a valuation to the formal parameters of procedure and the return value of A. The state machine rejects certain finite execution traces

[1] Not necessarily finite state.

(a sequence of interface states) of $P \mid\mid L$, either because P makes improper use of the API implemented by L or because L does not properly implement the API.

A program $Q = P \mid\mid L$ together with a SLIC specification S defines a new program Q' in which the specification S has been combined with Q. This is known as a "product construction" [22, 21]: Q' is the product of program Q with specification S. The product Q' has the following property: a unique label (`SLIC_ERROR`) is not reachable in Q' if-and-only-if Q satisfies specification S. We have developed a pre-processor for SLIC that produces the program Q' given inputs Q and S. There are two ways to use the program Q':

- One can subject Q' to a static analysis to attempt to determine whether or not `SLIC_ERROR` is reachable in Q' (an undecidable problem, of course), as in the SLAM project [2, 1].
- One can run Q' on various tests to see if execution ever reaches `SLIC_ERROR`.

The rest of this paper is organized as follows. Section 2 presents the syntax and semantics of SLIC. Section 3 describes how the SLIC pre-processor works. Section 4 shows how SLIC can be used to encode properties that involve universal quantification over dynamically-allocated objects in C. Section 5 reviews related work and Section 6 discusses future work.

2 Syntax and Semantics

This section presents the syntax and semantics of SLIC. The design for SLIC was driven by two desiderata:

- *Out-of-line specification*: we did not want to require programmers to modify or annotate source code, since in SLAM we are dealing mainly with legacy code;
- *C-like syntax/semantics*: we wanted SLIC specifications to resemble C code as much as possible.

2.1 Syntax

The syntax of the SLIC language is defined in Figure 1. A SLIC specification consists of two basic parts: a *state structure* definition and a list of *transfer functions*.

The state structure is a global C structure consisting of a set of fields. The type of a field can be (1) any scalar C type, or (2) a pointer to any C type. Structures, unions and arrays are not allowed. Each field must be given an initial value which defines the initial state of the state machine.

A transfer function has two parts: a pattern specification and a statement block that defines the function body. A pattern specification has two parts: a function identifier *id* and one of four basic event types (*event*): **call**, **return**, **entry**, **exit**. The first two events (**call** and **return**) identify the program points immediately before transfer of control to the named function takes place (just after evaluation of the actual expressions to the call), and immediately after control returns to the caller (just before assignment of the return value). The latter two events (**entry**

Syntax			Comment
S	::=	$state$ $transFun^+$	*A* SLIC *specification consists of a state structure, and a list of transfer function definitions.*
$state$	::=	**state** { $fieldDecl^+$ }	*A state structure is a list of field declarations.*
$fieldDecl$	::=	$ctype\ id = expr$;	*A field has a C type, an identifier and an initialization expression.*
$transFun$	::=	$pattern\ stmt$	*A transition function consists of a pattern and a statement.*
$pattern$	::=	id . $event$	
$event$	::=	**call** | **return** | **entry** | **exit**	
$stmt$	::=	$id^+ = expr^+$;	*Parallel assignment statement.*
	|	**if** ($choose$) $stmt$ [**else** $stmt$]	
	|	**abort** $string$**;**	
	|	**reset;**	
	|	**halt;**	
	|	{ $stmt$ }	
$choose$	::=	*	Non-deterministic choice
	|	$expr$	
$expr$	::=	id | $expr\ op\ expr$ | $\cdots$	*Pure expression sublanguage of C*
id	::=	$C\_identifier$	refer to fields of state structure
	|	**$** int	*$i refers to i^{th} actual/formal parameter*
	|	**$ return**	*return value of a function*
	|	**$** $C\_identifier$	*global variable*

Figure 1: Syntax of the SLIC language.

and **exit**) identify the program points in the named (called) function immediately before its first statement and immediately before it returns control to the caller.[2]

The body of a transfer function contains a single statement. A statement is either a C-style **if-then-else** conditional, a (parallel) assignment statement or **abort**, **halt** or **reset**. The **abort** statement is used to explicitly signal that an unsafe state has been reached with an error message. The **halt** statement signals that analysis of the current execution path should stop. The **reset** statement resets the values of all fields in the state structure to their initial values.

Two important control constructs are missing from the statement sublanguage: statement sequencing and loops. This restriction has several desirable consequences: (1) within the execution of a transfer function each state field can be updated at most once; (2) transfer functions always terminate.

The predicate in an **if-then-else** statement can either be the token "*****", which represents a non-deterministic boolean choice operator, or a boolean expression. Thus, a SLIC specification can encode non-deterministic state machines.

The expression sublanguage (*expr*) of SLIC is the pure expression language of C, without state update operators (`++`, `--`, etc.), pointer arithmetic, or the address-of operator (`&`). Deref-

[2]The reason for having four events instead of two is that sometimes the code for the library L may not be available for instrumentation. In this case, the **call** and **return** events must be used to monitor the program P calling functions in L, and the **entry** and **exit** events must be used to monitor the library L calling functions in P (via function pointers).

```
state {
  int zero_cnt = 0;
}

put.entry {
  if ($1 == 0) {
    if (zero_cnt == 4)
      abort "Queue has 4 zeroes!";
    else
      zero_cnt = zero_cnt + 1;
  }
}

get.exit {
   if ($return == 0)
     zero_cnt = zero_cnt - 1;
}
```

Figure 2: SLIC specification for a simple property of a global queue.

erencing via `*` and `->` is allowed. The identifiers in this language are of two forms: regular C-style identifiers that are (only) used to refer directly to the fields of the state structure. The $\$int$ identifiers are used to refer to the actual/formal parameters, depending on the event context (**call** and **entry** events).[3] The identifier **$return** is used to refer to the return value, which is accessible at the **exit** and **return** events. An identifier $\$C\_identifier$ is used to refer to global variables accessible in the API.

Figure 2 gives a very simple example of a SLIC specification for a global queue of integers. The specification states that it is in error to have more than four zeroes in the queue.

The state structure contains an integer to count the number of zeroes in the queue, initialized to zero (to reflect the initial empty state of the global queue). Upon entry to the `put` function of the queue (where the first parameter of the function is the integer), the transfer function checks to see if the parameter is zero. If it is not, then no state change occurs. Otherwise, the transfer function checks if the count of zeroes has reached four. If so, the transfer function aborts execution (that is, an unsafe state has been reached). Otherwise, the count is incremented. Upon exit from the queue's `get` function, if the return value of the function is zero then the state count is decremented.

2.2 Semantics

The semantics of a SLIC specification is straightforward and can be formalized using traces. An execution of a C program $Q = P \mid\mid L$ with a SLIC specification S can be seen as a *trace* of concrete execution states $\sigma_1 \rightarrow \sigma_2 \cdots$. An execution state $\sigma = (pc, \Omega)$ has two components: a program counter pc and a valuation Ω to all variables that are in scope at pc. The state

[3] Additionally, SLIC supports access to the parameter values at the **return** and **exit** events by caching the values at the corresponding **call** and **exit** events.

```
// C code generated from SLIC

static int zero_cnt = 0;

void SLIC_put_entry(int f1) {
  if (f1 == 0) {
    if (zero_cnt == 4)
      SLIC_ERROR("Queue has four zeroes!");
    else
      zero_cnt = zero_cnt + 1;
  }
}

void SLIC_get_return(int ret) {
   if (ret == 0)
     zero_cnt = zero_cnt - 1;
}
```

```
// queue (library) code, instrumented by SLIC

void put(int i) {
   SLIC_put_entry(i);
   ...
}

int get() {
   ...
   // computation into return temporary t
   ...
   SLIC_get_exit(t);
   return t;
}
```

Figure 3: SLIC generated code and instrumentation.

structure from S is added as an extra global variable to Q that is in scope at all values of *pc*. Each transition may cause a change in the state of the SLIC global state structure, depending on whether or not the program *pc* corresponds to one of the four event types that a SLIC specification matches on, and depending upon the code of the transfer function that may be invoked. An execution is an *unsafe execution* if it can lead to the execution of an **abort** statement.

3 Slic Instrumentation

Given a program $Q = P \mid\mid L$ and a SLIC specification S over L, the SLIC pre-processor tool creates a new program Q' as follows.

First, in order to simplify the instrumentation process, the SLIC pre-processor puts the C code into a normal form after parsing it. This normalization makes all control-flow explicit and makes all RHS expressions and argument expressions side-effect free. As a result of the normalization, a pretty printing of the SLIC-instrumented code will not look very similar to the input C code (although there is a direct mapping between the two).

The SLIC specification S then is compiled into a C file $S.c$ as follows. The SLIC state structure becomes a static global variable in $S.c$. Each transfer function t in S with name $n.e$ is compiled into a C function S_n_e with a formal parameter f_i for each unique $\$i$ (and $\$return$) referenced in the body of t. The body of f is compiled straightforwardly, with references to $\$i$ and $\$return$ variables replaced by references to the corresponding formal parameters defined above.

Finally, instrumentation of the program Q is accomplished by inserting a call to function S_n_e with appropriate actual arguments whenever the event $n.e$ takes place. In the presence of function pointers, without a points-to analysis, it is not (in general) possible to instrument for

```
state {
  enum  { Unlocked, Locked} s = Unlocked;
  T*    which_t               = NULL;
}

Allocate_T.return {
  if ($return && !which_t) {
     if (*) {
       which_t    = $return;
       s          = Unlocked;
     }
  }
}

Deallocate_T.call {
  if (which_t && $1 == which_t) {
    if (s == Locked)
      abort;
    which_t = NULL;
  }
}
```

```
Lock_T.call {
  if (which_t && $1 == which_t) {
    if (s == Unlocked)
      s = Locked;
    else
      abort;
  }
}

UnLock_T.call {
  if (which_t && $1 == which_t)
    s = Unlocked;
}
```

Figure 4: A SLIC specification of a locking protocol that encodes universal quantification over dynamically-allocated objects.

the **call** and **return** events. The control-flow normalization phase of SLIC uses the points-to analysis of Das [6] to resolve function pointers. Such an analysis is not needed if only **entry** and **exit** events are used.

Figure 3 shows the code generated from the specification of Figure 2 and some instrumented code.

4 The Universal Quantification "Trick"

In this section, we show how SLIC can be used to check safety properties that involve universal quantification over dynamically-allocated data in a C program. This relies on an "instrumentation trick" that uses non-determinism to express universal quantification. This trick has been used before in model checking [4].

Consider the following problem: library L allows a program to allocate, deallocate, lock and unlock objects of type `T*`. We wish to establish that a C program P that dynamically allocates and deallocates objects of type T follows a proper locking protocol for each object t of type T. More specifically, each object of type T begins in the "unlocked" state when successfully allocated by a call to the function `Allocate_T`. An object of type T may be locked (unlocked) by calling the function `Lock_T` (`Unlock_T`), passing the object to be locked as an argument. It is illegal to lock an object that already is locked. It is also illegal to deallocate (via `Deallocate_T`) a locked object.

Figure 4 shows how this property is encoded in SLIC. The state structure has two fields:

s represents the state of a "selected" object of type T. If *which_t* is not NULL then *which_t* contains the location of the selected object. Otherwise, no object has been selected. There are four transfer functions that update this state structure:

- `Allocate_T`.**return**: If an object has not been selected and the return value of this function is non-NULL (which means that an object of type T has been successfully allocated), then non-deterministic choice is used to determine whether or not this instance of type T will be "selected".
- `Deallocate_T`.**call**: If the object being deallocated is the "selected" object then the execution aborts if it is locked.
- `Lock_T`.**call**: If the object to be locked is the "selected" object then the execution aborts if already it is locked.
- `Unlock_T`.**call**: If the object to be unlocked is the "selected" object then the state is set to unlocked.

It should be noted that this "trick" is mainly helpful for static analysis, which can deal with a non-deterministic FSM as it has the power to explore all possible paths. On the other hand, at run-time the non-deterministic choice points in a SLIC specification must be resolved. It should also be noted that this trick is equivalent to checking the locking property for every static occurrence of `Allocate_T` in a program.

5 Related Work

5.1 Automata-based specification

From a research perspective, there is little new to SLIC. SLIC is a concrete realization of Schneider's security automata [21] for the C language.

Other automata-based specification languages for C have been used by Engler et al in the MC project [7] and Evans in the new version of LCLint [8]. Engler et al.'s Metal specification language has a more general event definition language, in which an arbitrary piece of C syntax can be recognized as an event. Additionally, Metal allows state to be attached to a pointer. While Metal is finite state and explicitly enumerates the possible states of the automaton, SLIC is not finite state (as it can count) and more expressive in this regard.

The run-time Java assurance tool Java-MaC uses a state machine approach based on events [14]. In this work, specification is separated into two parts: a primitive event definition language (PEDL) that generates a trace of events and a meta event definition language (MEDL) that implements a security automaton that checks the trace. MEDL contains history variables, as SLIC does.

5.2 Pre- and post- conditions

Another popular form of software specification is based on pre- and post-conditions [10] as found for languages such as Ada [17], Smalltalk and Larch [9, 3], Eiffel [18], and Java [16].

Although syntactically dissimilar, SLIC has equivalent power to such formalisms, as it can check arbitrary conditions on state at the entry and exit of functions. In some sense, SLIC is less "declarative" than a pure pre- and post- condition formalism as it has state, which is read from and written to. If enhanced with so-called "model variables", pre- and post-conditions can be used to specify temporal safety properties. In contrast to pre- and post-conditions, which annotate each function in an interface (splitting the definition of a property across function declarations), SLIC centralizes the definition of a partial property in a single state machine.

5.3 Temporal logics

Temporal logic has been used for specifying properties of software, notably in Holzmann's FeaVer system [12] (based on the SPIN model checker [11] and its use of Linear Temporal Logic) and the Bandera Specification Language [5]. The use of Linear Temporal Logic allows user to specify liveness as well as safety properties, while SLIC is restricted to safety properties.

5.4 Instrumentation languages

Rosenblum's APP [20] is an assertion pre-processing language and tool for C. Assertions are expressed in stylized comments in the C code and converted into instrumentation code by the APP tool. APP does not provide support for check safety properties, as it has no history mechanism.

SLIC is an example of so-called"aspect-oriented" programming [13], in which some behavior of a system is separated into an "aspect". A pre-processor then "weaves" together a program P and an aspect to generate a new program P' (a form of production construction). In general, an aspect can both read and modify the state visible to program P, while a SLIC specification can only read the state of P (a SLIC specification cannot modify the state visible to program P).

5.5 Code as specification

Given a program P accessing library L via an API, one can also specify properties by replacing L by a library L' that performs various checks on P. That is, a human can create the product program directly. This approach is used in IBM's CANVAS Java analysis project with the EASL/P language [19]. This language is a core subset of Java that allows the (abstract) specification of component behavior using Java code. It contains loops and the ability to allocate data dynamically and is strictly more powerful than SLIC.

6 Future Work

Besides obvious syntactic sugarings for convenience, SLIC can be extended in three basic ways. However, adding expressiveness to SLIC needs to be balanced with the ability to guarantee that (1) SLIC transfer functions always terminate, and (2) SLIC instrumentation does not alter the values of variables in the original program.

- *Events*: the alphabet of program events that SLIC recognizes can be enlarged beyond procedure call and return. Memory read and writes are obvious events to add first, which would allow SLIC to perform checks such as NULL pointer dereferencing. See the Metal language [7] for examples of such events.
- *State*: the domain of SLIC's state structure can be enlarged. In particular, there are examples where it would be useful for SLIC to have its own array, stack or other dynamic data structure in order to record more about the execution history.
- *Computation*: SLIC's computational ability can be enhanced by including procedures, loops, modules, objects, etc. We have found many cases where procedures would be helpful in giving more structure to SLIC transfer functions.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, LNCS 2057, pages 103–122. Springer-Verlag, 2001.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages (to appear)*. ACM, 2002.

[3] Y. Cheon and G. T. Leavens. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, July 1994.

[4] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *ICSE 00: Software Engineering*, 2000.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *SPIN 00: SPIN Workshop*, LNCS 1885. Springer-Verlag, 2000.

[6] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.

[7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*. Usenix Association, 2000.

[8] D. Evans. Static detection of dynamic memory errors. In *PLDI '96*, pages 44–53. ACM, May 1996.

[9] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[11] G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[12] G. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44:59 – 65, October 2001.

[14] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance tool for java programs. In K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[15] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[16] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.

[17] D. C. Luckham. Anna : A language for annotating Ada programs : A reference manual. In *LNCS 260*. Springer-Verlag, September 1987.

[18] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992.

[19] G. Ramalingam, A. Warshavsky, J. Field, and M. Sagiv. Deriving specialized heap analyses for verifying component-client conformance. Technical Report RC22145, IBM Research, August 2001.

[20] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[21] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[22] M. Y. Vardi and P. Wolper. An automata theoretic apporach to automatic program verification. In *LICS 86: Logic in Computer Science*, pages 332–344. IEEE Computer Society Press, 1996.