Runtime Verification of C Programs

Klaus Havelund

Jet Propulsion Laboratory
California Institute of Technology
Pasadena CA 91109, USA
Klaus.Havelund@jpl.nasa.gov

Abstract. We present in this paper a framework, RMOR, for monitoring the execution of C programs against state machines, expressed in a textual (non-graphical) format in files separate from the program. The state machine language has been inspired by a graphical state machine language RCAT recently developed at the Jet Propulsion Laboratory, as an alternative to using Linear Temporal Logic (LTL) for requirements capture. Transitions between states are labeled with abstract event names and Boolean expressions over such. The abstract events are connected to code fragments using an aspect-oriented pointcut language similar to ASPECTJ's or ASPECTC's pointcut language. The system is implemented in the C analysis and transformation package CIL, and is programmed in OCAML, the implementation language of CIL. The work is closely related to the notion of stateful aspects within aspect-oriented programming, where pointcut languages are extended with temporal assertions over the execution trace.

1 Introduction

The field of program verification is concerned with the problem of determining whether a program conforms to a specification. The pure verification problem consists of proving that all possible executions of the program conform to the specification. This is in general undecidable. Runtime verification is a less ambitious, but more feasible approach, just attempting to prove conformance of a single execution wrt. a specification. The specification can in this context be seen as a formalized oracle that can be used during testing, or it can become part of a fault protection system that runs in tandem with the program during its deployment, while triggering error correction code when non-conformance to the specification is detected.

The paper presents the runtime verification framework, RMOR (Requirement Monitoring and Recovery, pronounced *"armor"*), for monitoring C programs against state machines, using an aspect-oriented pointcut language to perform program instrumentation and connect the abstract events occurring in state machines with code fragments. The work has been partly driven by the context of embedded systems for planetary rovers and unmanned deep-space spacecraft as developed at NASA's Jet Propulsion Laboratory (JPL), where the majority of such code is written in C. The work presented reflects the following four observations. First, state machines appear a natural notation for programmers to apply, in contrast to for example temporal logic, or even regular expressions. Regular expressions are likely the most attractive of the succinct notations, but seem to be best suited for specifying "small" properties, whereas state machines support "big" properties involving many states. Second, although graphical editors for state machines are convenient, many programmers find textual programming-like notations convenient. Third, program instrumentation should be automated, connecting events to program points. Aspect-oriented programming has offered powerful pointcut languages for expressing such instrumentation. Fourth, most runtime verfication environments to date have been developed for Java, and C has been somewhat ignored. This is unfortunate since a majority of embedded software is written in C.

The RMOR language has inspirations from several sources. The language supports a notion of state machines directly influenced by RCAT (Requirement CApture Tool), a graphical state machine language language and editor [24, 25]. That graphical state machine language is inspired by Linear Temporal Logic (LTL) and allows for liveness properties to be stated as well as safety properties. This is achieved by introducing special *error states* and *liveness states*. RCAT was developed to support property specification for the SPIN model checker [17][1] and was together with RMOR products of the *Reliable Software Systems Development* (RSSD) project, funded by NASA. Beyond RCAT, another direct inspiration has been the STATL specification language [12], from where a distinction between *consuming* and *non-consuming* transitions was borrowed (a consuming transition leaves the source state, whereas a non-consuming leaves a "token" – does not consume the token – in the source state when the transition is taken). Finally aspect-oriented programming, and specifically ASPECTJ [18] has strongly inspired the pointcut language driving program instrumentation. More recently, ASPECTC [2] has emerged as an aspect-oriented framework for C. This will be discussed further in Section 7.

A considerable amount of research has been invested in program monitoring systems by different communities within the last 5-10 years. The runtime verification community is concerned with program correctness [10, 19, 13, 26, 11, 8]. This includes our own work [15, 16, 4]. Most of these efforts investigate more or less powerful temporal logics, with an exception in [11], which suggests the use of graphical UML state charts. The aspect-oriented programming community is investigating what is referred to as stateful aspects, where the pointcut language is extended with dynamic trace predicates [9, 29, 7, 28, 1]. These pieces of work are often extensions of ASPECTJ [18]. TRACEMATHCES [1] for example is an extension of ASPECTJ with regular expressions. JASCO [28] is a state machine solution for Java. An exception is ARACHNE [9], which performs runtime weaving into binary code of C programs. ARACHNE supports a form of trace predicates describing sequences of function calls, a limited form of regular expressions. The SLIC language [3] of the SLAM project is a specification language for C much resembling an aspect-oriented programming language, but simplified to support static verification as well as monitoring. The language supports state variables as well as access to function arguments and return values, but state machines have to be encoded using `enum` types, and the event language is not as comprehensive as a general purpose pointcut language. The program analysis communitiy has also contributed

[1] RCAT automata are by the RCAT tool translated into Büchi automata. RMOR can specifically monitor against such Büchi automata, although this is not the main purpose of the tool.

to this field [20] and the model checking community, which uses timed automata for testing, including monitoring [27, 6].

Our contributions to these efforts are: (i) to suggest a simple and natural textual programming notation for non-deterministic state machines integrated with an aspect-oriented pointcut language for program monitoring. This includes adapting the notions of error and live states from RCAT [24] for monitoring. With these concepts simple (finite trace) LTL properties can be stated naturally as state machines (the contribution of [24]) and monitored (our contribution). (ii) To implement such a system for C. Most embedded software is written in C. Most monitoring tools, however, have been focused on Java. The implementation uses CIL [21], which turns out to be very suited for developing source code instrumentation and runtime monitoring frameworks for C. (iii) To apply RMOR, resulting perhaps most importantly in feed-back from engineers wrt. usability.

The paper is organized as follows. Section 2 gives an overview of the RMOR architecture. Section 3 presents through examples the RMOR specification language. Section 4 summarizes the grammar of the specification language. Section 5 describes implementation details, including principles of the C code that is generated, as well as how the C code is instrumented. Section 6 presents case studies performed with RMOR. Finally Section 7 contains conclusions and outlines future work.

Acknowledgements Thanks goes to researchers at JPL. Margaret Smith designed the graphical RCAT language, which inspired the design of RMOR. Rajeev Joshi and Matthew Bennett participated in case studies using versions of RMOR. Gerard Holzmann supported the project by funding it as part of NASA's Reliable Software System Development project[2].

2 Overview of RMOR

The overall working of RMOR is illustrated in Figure 1. RMOR is a C program transformer, which inputs a pair consisting of a C program and a specification, and which outputs a C program that is *"armored"* by the specification. The specification is written in a textual format, that either can be programmed directly by a programmer, or it can be generated from a graphical state machine specification in the RCAT specification language. More specifically, RMOR takes as input a specification S in the RMOR specification language, and a C program P and produces a transformed program $Q = M + P_I$ which is the combination of a monitor M generated from the specification S, and an instrumented version P_I of P. P_I is P augmented with additional code that drives the monitor M. Executing the resulting program Q corresponds to executing the original program P, but with the monitor M constantly checking conformance to the specification. In case the specification is violated, an error message is printed on standard output, and in case specified, an error handling function is invoked.

[2] Part of the work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Part of the work was carried out at Kestrel Technology, under a contract with the National Aeronautics and Space Administration.

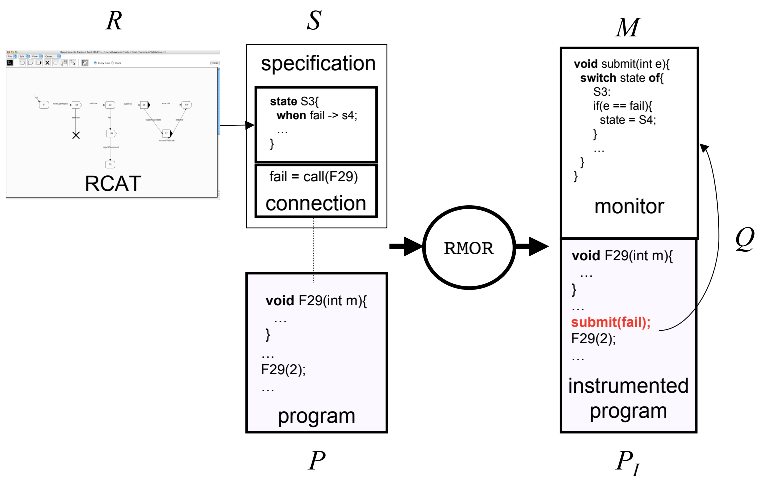

Fig. 1. Overview of RMOR

The specification consists of two parts: the behavioral specification expressed as a set of state machines, or monitors as they are called, and an instrumentation specification. The state machines contain states and transitions between states that are triggered by the occurrence of events. Events are just abstract names. The instrumentation part specifies how these abstract event names connect to the code and is the basis for the automated program instrumentation. In the resulting instrumented code P_I, calls to the monitor M occur as calls of the `M_submit(int event)` function. Events are represented as integers. The calls of this function are automatically inserted by RMOR at locations defined by the instrumentation specification. The monitor M itself is a set of synthesized C functions that check conformance to the state machines and which are written into an `rmor.c` file that has to be compiled and linked together with the application. An `rmor.h` header file is also generated that containts the events and RMOR API prototypes (function signatures). The header file does not need to be included in the user program under normal circumstances, but can be as explained later in case the user program needs to explicitly refer to monitoring functions. The synthesized monitor uses a fixed amount of memory, hence it does not use dynamic memory allocation.

RMOR is implemented using CIL (C Intermediate Language), a C program analysis and transformation system [21]. CIL is programmed in OCAML, which consequently also is the programming language in which RMOR is implemented. CIL is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs. The CIL tool parses a C program and generates an abstract syntax tree annotated with type information. The generated tree represents a program in a clean subset of C. CIL is very robust and has been applied to for example the Linux kernel and GCC's C torture testsuite and processes not only ANSI-C programs but also those using Microsoft C or GNU C extensions. Consequently RMOR inherits the same characteristics. CIL provides a driver which behaves as either the gcc or Microsoft VC compiler and can invoke the preprocessor followed by the CIL application. The advantage of this script is that one can easily use RMOR with existing make files. The RMOR system extends CIL with approximately 2500 lines of code.

3 The RMOR State Machine Language

3.1 An Example C Program

In order to illustrate the specification language, consider the following toy application program about which properties will be formulated. The program, located in a file `main.c`, defines a collection of functions supporting uplink of data from a planetary rover to a space craft[3]:

```
char* header;
Connection open_connection(char* name) {...}
bool close_connection(Connection connection) {...}
void cancel_transmission(Connection connection) {...}
void write_buffer(Connection connection, int data) {...}
void commit_buffer(Connection connection) {...}
void acknowledge() {...}
void debug(char* str){...}

main(){
  Connection c1,c2;
  c1 = open_connection("connection1");
  c2 = open_connection("connection2");
  write_buffer(c1,100);
  commit_buffer(c1);
  close_connection(c1);
}
```

The program offers functions for opening and closing a connection between rover and space craft. While the connection is open data can be written to a data buffer, and finally committed, for which an acknowledgment is received and recorded with a call of the function `acknowledge`. A transmission can also be cancelled, not requiring further action. The program contains a global variable `header`, containing information about the current connection. The main program illustrates an example scenario.

[3] The example is fiction and does not represent an existing design.

3.2 Writing a Monitor

RMOR allows to specify properties about the execution order of *function calls* and *global variable accesses*. RMOR monitors safety properties, usually formulated as "*nothing bad happens*", as well as termination-bounded liveness properties "*something good eventually happens before program termination*". Safety properties are checked each time an event is submitted. Liveness properties are checked at the end of an execution when monitoring is terminated: at that point it is checked whether any outstanding events have not happened that were expected to happen according to the requirements represented by the monitors. In order to illustrate the RMOR notation a set of requirements will be modeled. Consider the following requirements R_1, R_2 and R_3 about the call-sequence of the functions in the above API. R_1*: "A connection is opened, accessed zero or more times, and subsequently either closed or canceled. An access is either a write operation or a commit operation";* R_2*: "The commit operation must be followed by an acknowledgement before any other operation can be performed, except a cancellation";* R_3*: "It is illegal to have more than one connection opened at any time".* These requirements can be formulated as several monitors, for example one for each requirement, or they can be grouped into one monitor as follows.

```
monitor UplinkRequirements {
  event OPEN = after call(main.c:open_connection);
  event WRITE = after call(main.c:write_buffer);
  event COMMIT = after call(main.c:commit_buffer);
  event ACK = after call(main.c:acknowledge);
  event CANCEL = after call(main.c:cancel_transmission);
  event CLOSE = after call(main.c:close_connection);

  initial state Closed {
     when OPEN -> Opened;
     when WRITE || COMMIT || ACK || CLOSE => error;
  }

  live state Opened {
    when COMMIT -> Committing;
    when CLOSE -> Closed;
    when ACK => error;
  }

  next state Committing {
    when ACK -> Opened;
  }

  super Active[Opened,Committing]{
    when CANCEL -> Closed;
    when OPEN => error;
  }
}
```

The monitor introduces six events to be monitored and a state machine that any event sequence observed during program execution must conform to. Each event is defined by a predicate, denoting a set of statements in the program that satisfies it (a pointcut using aspect-oriented terminology), and a directive indicating whether the event should emitted before or after any statement satisfying the pointcut. As an example, the event OPEN is associated with the pointcut `call(main.c:open_connection)` which is matched by any *call* of the function `open_connection` defined in the file `main.c`. In the example program there are in fact two such calls. The `after` directive requires the event to be emitted to the monitor *after* each of these calls. It is in essence an instruction to RMOR to instrument the code by inserting a call to the monitor after these two calls. Similarly for the other events. Note that following aspect-oriented ideas, the program is oblivious to the fact that it is getting instrumented.

The state machine itself consists of three basic states: `Closed`, `Opened` and `Committing`. Each state is modeled as a named (the name of the state) block enclosed by curly-brackets `{ ... }` containing all its exiting transitions. The `Closed` state is the initial state, indicated with the *state modifier* keyword `initial`. In the `Closed` state, two transitions are defined. The first transition states that the event `OPEN` brings the monitor into the `Opened` state. Recall that an `OPEN` event occurs after any call of the function `main.c:open_connection`; The second transition states that if any of the events in the set {`WRITE`, `COMMIT`, `ACK`, `CLOSE`} occurs, using the *or*-operator '`||`', it is it is regarded as an error – `error` is a special identifier denoting a built-in error state. The double arrow (`=>`) indicates a transition that leaves a token in the source state, in this case `Closed`, such that also future violations of this property is detected. Such a transition is called *non-consuming* since it does not consume the source token, as does the normal single arrow *consuming* transition (`->`). Recall that state machines are non-deterministic.

The `Opened` state is a *live* state as indicated by the modifier keyword `live`, meaning that this state must be left before program termination for this specification to be satisfied. This specifically means that either a `COMMIT` event or a `CLOSE` event must occur. An `ACK` event is not allowed to occur in this state. In the `Committing` state an `ACK` event must occur as the *next* observable event, indicated by the `next` state modifier keyword. This has as consequence that no other event can occur, except for a cancellation. The latter exception is a consequence of the super state named `Active` defined at the end of the monitor. This super state contains the two atomic states `Opened` and `Committing` and has two exiting transitions. This is a shorthand for these exiting transitions connected to each substate. The super state definition implies that when in any of the two sub-states it is regarded as an error if an `OPEN` event occurs, and a `CANCEL` event brings the monitor back to the initial `Closed` state.

3.3 Complex Pointcuts

Emissions of events to a monitor are inserted either before or after certain program locations (joinpoints) identified by pointcut expressions occurring after the '`=`' sign in event definitions. Pointcut expressions can, similar to ASPECTJ and ASPECTC [2], be used directly in event definitions, as we have seen above, or they can be defined and given names in explicit pointcut declarations, using Boolean combinators similar to those used on conditions. The following example illustrates this. Consider the additional requirement R_4*: "A write operation or an assignment to the* `header` *variable (collectively referred to as an update) should be followed by a commit operation before the connection is closed, unless the transmission is cancelled. This, however, only concerns* main updates *performed in the* `main.c` *file, ignoring updates made within any debugging function"*. In order to capture this requirement RMOR's poincut language is used to define the notion of a *main update*. The following monitor defines two poincuts, one used to define the other, and an event that is defined in terms of the latter pointcut.

```
monitor Symbols {
  pointcut Update = call(main.c:write*) || set(main.c:header);
  pointcut MainUpdate = Update &&
                      within(main.c) && !withincode(*debug*);

  event UPDATE = after MainUpdate;
}
```

The pointcut `Update` matches any program statement that is either: (*i*) is a call of a function defined in `main.c` and with a name matching the pattern `write*`, meaning having the name `write` as prefix, or (*ii*) is an update of the variable `header` declared in `main.c`. The file patterns (the part before and including the '`:`') are optional. Both file names and function/variable names can be indicated as patterns using "`*`" to represent any sequence of symbols. The pointcut `MainUpdate` refines the first pointcut to only concern those program statements occurring in the file `main.c` but not within any function with a name that contains the string `debug`. Finally, the event `UPDATE` is emitted after any main update. Note that this monitor contains no state machine and is purely introduced to define the pointcuts and the event. RMOR allows a monitor to import other monitors to access their pointcuts and events, and the next monitor imports the just presented one to access the `UPDATE` event, and also imports the original monitor to access further events.

```
monitor CommitUpdates {
  import Symbols;             // access UPDATE
  import UplinkRequirements; // access COMMIT, CANCEL and CLOSE

  state Start {
    when UPDATE => DoCommit;
  }

  live state DoCommit {
    when COMMIT -> Done;
    when CANCEL -> Done;
    when CLOSE -> error;
  }

  state Done{}
}
```

3.4 Error Handling

Our example program violates requirements R_2 (*a commit must be followed by an acknowledgment before anything else*), and R_3 (*no more than one open connection at a time*). Running the armored program produced by RMOR therefore causes two error messages to be printed on standard output. It is possible to provide a call-back handler function, which the monitor will call for each violation detected. This function must have the following name and type:

```
void handler(char *monitor, char *state, int kind) {
  ... user defined code ...
}
```

The first argument indicates the name (a string) of the monitor in which the error was encountered. The second argument indicates the name of the state it occurred in, and finally the third argument indicates the kind of error (a number between 0 and 2): (0) transition into an *error* state, (1) not leaving the *next* state before another event occurs, and (2) terminating in a *live* state. In order for errors to be handled by the handler function, the monitor must be declared with the `handled` modifier as follows:

```
handled monitor UplinkRequirements {
  ... as before ...
}
```

4 Elements of the RMOR Grammar

In this subsection elements of the grammar of RMOR are outlined, summarizing the concepts introduced in the example. A specification consists of a sequence of monitors[4]:

```
<specification> ::= <monitor>*
<monitor> ::=
      "handled"? "monitor" <monitor_name> "{" <declaration>* "}"
<declaration> ::=
      <import_decl> | <pointcut_decl> | <event_decl> |
      <state_decl> | <machine_decl>
```

An import declaration has the form:

```
<import-decl> ::= "import" <ident> ";"
```

Imports have the sole purpose of giving access to pointcuts and events from other monitors. Imports have no semantics at the state machine level. The grammar rules for pointcut declarations and pointcut expressions are as follows:

[4] The meta symbol `*` means zero or more occurrences, and `?` means zero or one occurrence.

```
<pointcut_decl> ::= "pointcut" <ident> "=" <pointcut_expr> ";"
<pointcut_expr> ::=
    "call" "(" (<idpat1>":")?<idpat2> ")"
  | "set" "(" (<idpat1>":")?<idpat2> ")"
  | "within" "(" <idpat1> ")"
  | "withincode" "(" (<idpat1>":")?<idpat2> ")"
  | <ident>
  | <pointcut_expr> "&&" <pointcut_expr>
  | <pointcut_expr> "||" <pointcut_expr>
  | "!" <pointcut_expr>
  | "(" <pointcut_expr> ")"
<idpat1> ::= ("*" | letter|digit | "_" | "." | "-" | "/" )+
<idpat2> ::= ("*" | letter|digit | "_" )+
```

A poincut expression can specify a function call or a variable assignment, with `idpat1` indicating the name of the file in which the called function or updated variable is declared. The within pointcut matches statements occurring in files with names matching the argument, and withincode matches statements occurring within functions with names matching the argument. Beyond this, pointcuts can be referred to by name and conjoined with Boolean operators. An event declaration has one of two forms:

```
<event_decl> ::=
    "event" <ident> "=" ("before "|" after") <pointcut_expr> ";"
  | "event" <ident> ("," <ident>) ";"
```

The event declarations shown this far are all of the first form. The second form is an abstract event declaration. It just introduces an event name that then can be used in state machines. However, no automated instrumentation is performed and it is the responsibility of the user to manually instrument the program to emit these events using the RMOR API. A state declaration can be of one of two forms:

```
<state_decl> ::=
    <state_modifier>* "state" <ident> "{" <transition>*  "}"
  | "super" <ident> "[" <ident> ("," <ident>)* "]" "{"
       <transition>*
    "}"
<state_modifier> ::= "initial" | "anytime" | "live" | "next"
<transition> ::= "when" <cond> ("->"|"=>") <ident> ";"
<cond> ::=
    "ANY" | <ident> | "!"<cond> | "(" <cond> ")"
  | <cond> "&&" <cond> | <cond> "||" <cond>
```

The first form is the basic state definition: a list of state modifiers, the keyword `state`, the name of the state, and a list of exiting transitions enclosed in a block. The second form is a super state definition, with the name of the super state and the list of sub-states in between `[ ... ]` brackets. These sub-states must be defined within the same monitor using the first form of state declaration. It is not possible to use another super state as a sub-state. The super state also has a list of exiting transitions. An `anytime` state always contains a token, even if an exiting transition is taken (state machines can be non-deterministic). The same effect can be obtained by defining all exiting transitions as non-consuming using the => arrow. A condition is a Boolean expression over event identifiers and the `ANY` keyword, which in essence represents `true`, or ”any” transition.

In an attempt to offer the possibility of grouping together state machines in one module it has been made possible to define several state machines inside a monitor. Such state machines cannot define any symbols or perform any imports:

```
<machine_decl> ::= "machine" <ident> "{" <state_decl>* "}"
```

RMOR offers in addition an API of functions with which the user application can interact with the monitors. These functions can for example be called from the handler. This includes functions for resetting and stopping monitors, submitting events, and printing monitor status for debugging purposes.

5 Implementation

OCAML [22] and its parser modules OCAMLLEX and OCAMLYACC were used to implement the parser for the RMOR specification language. The generated monitors in C utilize the SGLIB library [23], specifically double-linked lists for implementing sets. The program instrumentation module was, as already mentioned, implemented in OCAML on top of CIL [21].

5.1 Monitor Generation

The lexical scanning of RMOR specifications involves scanning of pointcut expressions, which is a well-known problem in aspect-oriented programming implementations, requiring the lexer to be state oriented, behaving differently in the *normal* and the *pointcut* state. OCAMLLEX allows for such state orientation, permitting us to apply a high-level parser generator for the task[5]. The program is parsed into an abstract syntax tree (AST), which is then processed for two purposes: translation of state machines to monitors, and instrumentation of the C code to emit events to the monitors (Section 5.2). The translator that produces state machines takes the AST as input and prints out the monitors in the file `rmor.c`. There are three constraints that specifically influence how RMOR is implemented: (i) monitors are allowed to be non-deterministic (a consequence for example of the => transition arrow, useful for monitoring), meaning that a state machine can be in more than one state at a given moment; (ii) dynamic memory allocation is not allowed since monitors should be able to monitor embedded flight code as part of a fault protection strategy, where only static memory allocation is allowed; (iii) a future extension of RMOR should allow for events to be parameterized with data values, and hence tokens in states should be able to carry values.

The first constraint requires each transition to produce a **set** of *next states*, computed from the set of *current states*. The second constraint requires that these different sets

[5] The ASPECTJ parser is for example not constructed using a parser generator.

cannot be allocated dynamically on the fly as new sets are built. Instead, all states are allocated up front, and for each monitor is maintained three collections during next-state computation: a list of *free states*, a set of *current states*, and a set of *next states*. Each collection is modeled as a double-linked list. All states are initially stored in the free list. The monitor subsequently just moves states between these three sets when a new event arrives. At program termination it is checked that no tokens exist in live or next states. The motivation for representing sets as linked lists of records, and not as bitvectors, is the third constraint above, which requires data values to be part of state tokens in an extension of the tool. This will be further discussed in Section 7.

5.2 Instrumentation with CIL

The instrumentation module is implemented using CIL's object oriented visitor pattern framework. RMOR defines a class that subclasses a predefined visitor class, overriding a method for each kind of CIL construct that should be visited. CIL's visiting engine scans depth-first the structure of a CIL program and at each node executes the corresponding method in the user-defined visitor. The code below shows part of the visitor class defined for instrumentation. It overrides the method `vinst : instr -> instr list visitAction` that is applied to every basic instruction in the C program (essentially function calls and assignment statements, excluding composite statements, such as loops). This function is expected to return a list of instructions, namely those that the visited instruction is replaced with. The body of the function computes a list of advices to be inserted (`advice_inserts`), that if not empty is split into those to be inserted before and after the instruction respectively.

```
class instrumentVisitor = object (self) inherit nopCilVisitor
  ...
  method vinst (i : instr) : instr list visitAction =
   ...
     let advice_inserts = match_instr ... i in
     if advice_inserts = [] then
       SkipChildren
     else
     begin
       let (before,after) = create_before_after advice_inserts in
       ChangeTo (before @ [i] @ after)
     end
end
```

The instrumentation consists of inserting calls of the function `M_submit(int event)` before or after joinpoints matching the pointcuts associated with events. The function `M_submit` stores the submitted event for later reference in the state machines, and subsequently calls the *next-state* function of each state machine.

6 Case Study

The Laboratory for Reliable Software at JPL has been developing a RAM File System (RAMFS) for use in future space missions. RAMFS will specifically be used as a backup file system on the next Mars Rover, MSL (Mars Science Laboratory), with launch date September-October 2009. MSL will be the biggest rover yet sent to Mars, and will be three times as heavy and twice the width of the Mars Exploration Rovers (MERs) that landed in 2004. RAMFS implements a thread-safe file system for flight systems in volatile memory (memory that requires a power supply to maintain the stored information). The main purpose of RAMFS is to provide a storage capability that can be used when the disk- or flash-file system is unavailable, e.g., when a spacecraft is in crippled mode, or in case there is not enough disk memory available. It is a project goal to apply various testing and verification technologies to establish confidence in the correctness of this file system [14]. Two different properties were formulated in RMOR and checked against the system. Both properties were satisfied, and malicious manual code modification caused them to be violated as expected.

Property 1: Matching Semaphore Accesses The first property, called MatchSem, checks that semaphore operations are executed correctly: the semaphore must be reserved and released in strictly alternating order. The specification further states that once the semaphore has been reserved, it must eventually be released again. Reserving and releasing the semaphore is performed in the program respectively by calls of functions `osal_sem_take` and `osal_sem_give`. The monitor is defined as follows.

```
monitor MatchSem {
  event semtake = before call(osal_sem_take);
  event semgive = after call(osal_sem_give);

  state Start {
    when semtake -> HaveLock;
    when semgive -> error;
  }

  live state HaveLock {
    when semgive -> Start;
    when semtake -> error;
  }
}
```

Property 2: Protected Memory Updates While the first property above states that the semaphore is used correctly, the second property states that memory accesses are correctly protected by the semaphore. That is, any access to memory must occur between a `semtake` and a `semgive`. Memory accesses come in two forms. The first are updates to the list of free memory through memory allocations with the function `ramfs_alloc_pages`, and memory freeing with the function `ramfs_free_pages`. The second are updates to the memory pages themselves through two functions `ramfs_update_entry` and `ramfs_update_header`. The monitor defines two pointcuts `free_list_update` and `page_update`, corresponding to these two kinds of calls.

```
monitor DataProtected {
  import MatchSem ;
  pointcut free_list_update =
     call(ramfs_alloc_pages) || call(ramfs_free_pages);
  pointcut page_update =
     call(ramfs_update_entry) || call(ramfs_update_header);
  event update = before free_list_update || page_update;

  state Unsafe {
    when semtake -> Safe;
    when update -> error;
  }

  state Safe {
    when semgive -> Unsafe;
  }
}
```

Observations This case study demonstrated the ease with which a non-expert in RMOR was able to quickly learn the specification language and formulate properties. Although not seen as a limitation during the exercise, the need for events to carry data values comes to the forefront in this example, specifically when it comes to the first property, that *the semaphore must be reserved and released in strictly alternating order*. The specification should ideally state that for a given semaphore *S*, its acquisition should be followed by a release of this same *S*.

A different case study was performed using RCAT from which RMOR monitors were generated as Büchi automata for an earlier version of RMOR. The case study was a rover controller for the Rocky 8 rover, a research vehicle that is used at JPL to develop, integrate, and demonstrate advanced autonomous robotic capabilities for future Mars missions. Since the specification language used was RCAT and since monitors were generated for an earlier version of RMOR, we shall not provide details about the example or the specifications. It suffices to say that the specification concepts used were similar to those of RMOR, and that the study supported the need for augmenting RMOR with the ability to express time constraints, and the ability to model conditions (predicates) on the state of the C program and use these as guards on transitions.

Concerning efficiency, the overhead naturally depends on the ratio with which monitored function calls and variable accesses are performed in the monitored application compared with the overall computation. Experiments showed that a single monitored call of a function with empty body results in an order of magnitude slow down of that call. Although monitored function calls usually constitute a small fraction of the overall computation, such overhead must be reduced using static analysis and algorithm optimizations.

7 Conclusions and Future Work

The following three aspects are important for acceptance of a technology such as RMOR: (i) *convenience* of the specification language; (ii) *expressiveness* of the specification language; (iii) *efficiency* of monitoring. A contribution of the paper is to illustrate the convenience of a state machine notation in combination with an aspect-oriented pointcut language. Concerning expressive power of the specification language, it currently only offers monitoring of propositional events. The notation should be extended with the ability to parameterize events with data values, corresponding to arguments in monitored functions, timers, and to generally enable C code to occur in the specification, for example allowing C code to be executed as a result of state machine transitions. In current work we are permitting this by directly extending ASPECTC [2] with state machines, utilizing ASPECTC's already existing pointcut language. This work is carried out using the SILVER extensible compiler framework [30]. Future work includes allowing user defined temporal logic operators as shorthands for state machines. Specifically, we plan to allow monitors to be parameterized with pointcuts. This will allow to define temporal operators/specification patterns within the language as is done in the EAGLE specification language [4], permitting very succinct specifications. We are furthermore exploring the possibility of adopting the more expressive rule-based logic RULER [5] as core logic, in which state machines form a special case. Efficiency can be obtained by application of static analysis to reduce code instrumentation.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA'05*. ACM Press, 2005.
2. AspectC. `http://research.msrg.utoronto.ca/ACC`.
3. T. Ball and S. K. Rajamani. SLIC: a Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2001.
4. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In B. Steffen and G. Levi, editors, *Proc. of Fifth International VMCAI conference (VMCAI'04)*, volume 2937 of *LNCS*. Springer, January 2004.
5. H. Barringer, D. Rydeheard, and K. Havelund. Rule Systems for Run-Time Monitoring: from Eagle to RuleR. In *Proc. of the 7th International Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, Vancouver, Canada, 2007. Springer.
6. S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis. Testing Conformance of Real-Time Applications by Automatic Generation of Observers. In *Proc. of the 4th International Workshop on Runtime Verification (RV'04)*, volume 113 of *ENTCS*. Elsevier, 2004.
7. C. Bockisch, M. Mezini, and K. Ostermann. Quantifying over Dynamic Properties of Program Execution. In *2nd Dynamic Aspects Workshop (DAW05), Technical Report 05.01*, pages 71–75. Research Institute for Advanced Computer Science, 2005.
8. F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, 2007.
9. R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Sgura-Devillechaise, and M. Südholt. An Expressive Aspect Language for System Applications with Arachne. In *Proc. of the 4th international conference on Aspect-oriented software development*, Chicago, USA, March 2005. ACM Press.

10. D. Drusinsky. Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions. In *Proc. of the 4th International Workshop on Runtime Verification (RV'04)*, volume 113 of *ENTCS*, Barcelona, Spain, 2004. Elsevier.
11. D. Drusinsky. *Modeling and Verification using UML Statecharts*. Elsevier, 2006. ISBN-13: 978-0-7506-7949-7, 400 pages.
12. S. Eckmann, G. Vigna, and R. A. Kemmerer. STATL Definition. Reliable Software Group, Department of Computer Science, University of California, Santa Barbara, CA 93106, 2001.
13. B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proc. of the 1st International Workshop on Runtime Verification (RV'01)*, volume 55(2) of *ENTCS*. Elsevier, 2001.
14. A. Groce and R. Joshi. Extending Model Checking with Dynamic Analysis. In F. Logozzo, D. Peled, and L. Zuck, editors, *Proc. of Ninth International VMCAI conference (VMCAI'08)*, LNCS. Springer, January 2008.
15. K. Havelund and G. Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, 24(2), March 2004.
16. K. Havelund and G. Roşu. Efficient Monitoring of Safety Properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.
17. G. J. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, 2004.
18. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.
19. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proc. of the 1st International Workshop on Runtime Verification (RV'01)*, volume 55(2) of *ENTCS*. Elsevier, 2001.
20. M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors using PQL: a Program Query Language. In *Proc. of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM Press, 2005.
21. G. C. Necula, S. McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proc. of Conference on Compilier Construction*, 2002.
22. OCAML. `http://caml.inria.fr/index.en.html`.
23. SGLIB. A Simple Generic Library for C. `http://sglib.sourceforge.net`.
24. M. Smith. Requirements for the Demonstration Version of the Requirements Capture Tool (RCAT). JPL/RSS Technical Report, RSS Document Number: ESS-02-001, 2005.
25. M. Smith and K. Havelund. Requirements Capture with RCAT. Jet Propulsion Laboratory, California Institute of Technology. Submitted for publication, February 2008.
26. V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Proc. of the 5th International Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*. Elsevier, 2005.
27. T-UPPAAL. `http://www.cs.aau.dk/~marius/tuppaal`.
28. W. Vanderperren, D. Suvé, M. Augustina Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. In *4th International Workshop on Software Composition, ETAPS 2005*, volume 3628 of *LNCS*. Springer, 2005.
29. R. Walker and K. Viggers. Implementing Protocols via Declarative Event Patterns. In R.N. Taylor and M.B. Dwyer, editors, *ACM Sigsoft 12th International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159–169. ACM Press, 2004.
30. E. V. Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an Extensible Attribute Grammar System. In *Workshop on Language Descriptions, Tools, and Applications*, 2007.