Interactive Runtime Verification - When Interactive Debugging meets Runtime Verification

Raphaël Jakse, Yliès Falcone, Jean-François Méhaut and Kevin Pouget
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble France
FirstName.LastName@univ-grenoble-alpes.fr

***Abstract*—Runtime Verification consists in studying a system at runtime, looking for input and output events to discover, check or enforce behavioral properties. Interactive debugging consists in studying a system at runtime in order to discover and understand its bugs and fix them, inspecting interactively its internal state.**

Interactive Runtime Verification (i-RV) combines runtime verification and interactive debugging. We define an efficient and convenient way to check behavioral properties automatically on a program using a debugger. We aim at helping bug discovery and understanding by guiding classical interactive debugging techniques using runtime verification.

I. Introduction

When developing software, detecting and fixing bugs as early as possible is important. This can be difficult: an error does not systematically lead to a crash, it can remain undetected during the development. Besides, when detected, a bug can be hard to understand, especially if the method of detection does not provide methods to study the bug.

a) Interactive debugging: A widespread way to fixing bugs consists in observing a bad behavior and starting a debugging session to find the cause. A debugging session generally consists in repeating the following steps: executing the program in a debugger, setting breakpoints before the expected cause of the bug, finding the point in the execution where it starts being erratic and inspecting the internal state (callstack, values of variables) to determine the cause of the problem. The program is seen as a white box and its execution as a sequence of program states that the developer inspects step by step using a debugger in order to understand the cause of a misbehavior. The execution is seen at a low level (assembly code, often mapped to the source code) while one would ideally want it be abstracted. The debugger links the binary code to the programming language. The state of the program can be modified at runtime: variables can be edited, functions can be called, the execution can be restored to a previous state. This lets the developer test hypotheses on a bug without having to modify the code, recompile and rerun the whole program, which would be time consuming. However, this process can be tedious and prone to a lot of trials and errors. Moreover, observing a bug does not guarantee that this bug will appear during the debugging session, especially if the misbehavior is caused by a race condition or a special input that was not recorded when the bug was observed. Interactive debugging does not target bug discovery: usually, a developer already knows the bug existence and tries to understand it.

b) Monitoring: Runtime verification (aka monitoring) [1], [2], [3] aims at detecting bugs. The execution is abstracted into a sequence of events of program-state updates. Monitoring aims at detecting misbehaviors of a black-box system: its internal behavior is not accessible and its internal state generally cannot be altered. Information on the internal state can be retrieved by instrumenting the execution of the program. The execution trace can be analyzed offline (i.e. after the termination of the program) as well as online (i.e. during the execution) and constitutes a convenient abstraction on which it is possible to express runtime properties.

We aim at easing *bug discovery*, *bug understanding* as well as their *combination*. We introduce Interactive Runtime Verification (i-RV), a method that brings bug discovery and bug understanding together by combining *interactive debugging* and *monitoring*. i-RV gathers strong points of both approaches by augmenting debuggers with runtime verification techniques. Using i-RV, one can discover a bug and start getting insight on its cause at the same time. i-RV aims at *automating and easing (manual) traditional interactive debugging* during the development. For instance, it is possible to automatically stop the execution when a misbehavior is detected or to automate checkpointing at the right times. We define an expressive *property model* that allows flexibility when writing properties. We give a *formal description* of our execution model using high-level pseudo-code which serves as a *basis for a solid implementation* and reasoning and to ensure correctness of our approach. End-users are however not required to have a full understanding of this description. i-RV takes advantage of *checkpoints*. Checkpoints allow saving and restoring the program state. They are a powerful tool to explore the behavior of programs by trying different execution paths. i-RV introduces the notion of *Scenarios*. They allow defining actions that are triggered depending on the current state of the property verification. We provide a *full-featured tool* for i-RV, Verde, written in Python as a GDB extension, facilitating its integration to developers' traditional environment. Verde also provides an optional *animated view* of the current state of the monitor. We give a *detailed evaluation* of i-RV using Verde. This evaluation validates the usefulness of i-RV and its applicability in terms of performances. An extended version of this paper is available [4] and Verde can be found at [5].

II. Approach Overview

In i-RV, the developer provides a property to check against the execution trace of a program to debug. The property can be written according to its specification or the Application Programming Interface (API) of the libraries it uses. An example of a property is pictured in Fig. 1 and gives the verdict false as soon as a queue q overflows. The program is run with

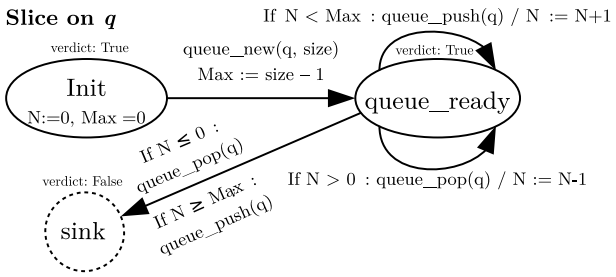

Figure 1. Property for the absence of overflow for each queue q in a program.

a debugger which provides tools to instrument its execution, mainly breakpoints and watchpoints, and let us generate events to build the trace, including function calls and variable accesses. An extension of the debugger provides a monitor that checks this property in real time. Breakpoints and watchpoints are automatically set at relevant locations as the evaluation of a property requires monitoring function calls and memory accesses. When an event stops influencing the evaluation of any property, the corresponding instrumentation (breakpoints, watchpoints) becomes useless and is therefore removed: the instrumentation is *dynamic*. The user-provided scenario defines what actions should be taken during the execution according to the evaluation of the property. Examples of scenarios are: when the verdict given by the monitor becomes false (e.g. when the queue overflows), the execution is suspended to let the developer inspect and debug the program in the usual way, interactively; save the current state of the program (e.g. using a checkpoint, a feature provided by the debugger) while the property holds (e.g. while the queue has not overflown) and restore this state later, when the property does not hold anymore (e.g. at the moment the queue overflows). When an event is generated — when a breakpoint or a watchpoint is reached — at runtime, the monitor updates its state. Monitor updates are seen as input events for the scenario. Examples of these events are "the monitor enters state X", "the state X has been left", "an accepting state has been entered", "a non-accepting state has been left".

III. Related Work

i-RV is related to several families of techniques for finding and fixing bugs.

a) Interactive and Reverse Debugging: Tools used in interactive debugging are mainly debuggers such as GDB, LLDB and the Visual Studio debugger. Reverse debugging [6], [7], [8] is a complementary debugging technique. A first execution of the program showing the bug is recorded. Then, the execution can be replayed and reversed in a deterministic way, guaranteeing that the bug is observed and the same behavior is reproduced in each replay. UndoDB and rr are GDB-based tools allowing record and replay and reverse debugging with a small overhead. i-RV also allows to restore the execution in a previous state using checkpoints, with the help of the monitor and the scenario, adding a level of automation.

b) Static Analysis and Abstract Interpretation: With heavyweight verification techniques [9], the source code of the software is analyzed without being run in order to find issues. Properties can also be proven over the behavior of the software. Unfortunately, theses approaches can be slow, limited to certain classes of bugs or safety properties and can produce false positives and false negatives. SLAM is based on static analysis and aims at checking good API usage. SLAM is restricted to system code, mainly Windows [10], [11] drivers.

c) Dynamic Binary Instrumentation: DBI allows detecting cache-related performance and memory usage related problems. The monitored program is instrumented by dynamically adding instructions to its binary code at runtime and run in an virtual machine-like environment. Valgrind [12] is a tool that leverages DBI and can interface with GDB. It provides a way to detect memory-related defects. DBI provides a more comprehensive detection of memory-related defects than using the instrumentation tools provided by the debugger. However, it is also less efficient and implies greater overheads when looking for particular defects like memory leaks caused by the lack of a call to the function `free`.

d) Instrumentation Based on the VM of the Language: For some languages like Java, the Virtual Machine provides introspection and features like aspects [13], [14] used to capture events. This is different from our model which rather depends on the features of the debugger. JavaMOP [15] is a tool that allows monitoring Java programs. However, it is not designed for inspecting their internal state. JavaMOP also implements trace slicing as described in [16]. In our work, events are dispatched in slices in a similar way, We do not implement all the concepts defined by [16] but this is sufficient for our purpose. In monitoring, the execution of the program can also be affected by modifying the sequence of input or output events to make it comply with some properties [17]. This is different from i-RV which applies earlier in the development cycle. We rather modify the execution from inside and fix the program than its observable behavior from outside.

e) Debugger-Based Instrumentation: Morphine, Opium and Coca [18] are three automated trace analyzers. The analyzed program is run in another process than the monitor, like in our approach. The monitor is connected to a tracer. Like in our approach, this work relies on the debugger to generate events. Focus is set on trace analysis: interactivity is not targeted and the execution remains unaffected.

f) Frama-C [19]: Frama-C is a modular platform aiming at analyzing source code written in C and provides plugins for static analysis, abstract interpretation, deductive verification, testing and monitoring programs. It is a comprehensive platform for verification. It does not support interactive debugging nor programs written in other programming languages.

Current approaches to finding and studying bugs have their own drawbacks and benefits and are suitable for discovering different sorts of bugs in different situations. Their relevance is also related to a phase of the program life cycle. None of them gather bug discovery and understanding.

IV. Joint Execution of the Debugger, the Monitor and the Program

i-RV relies on the joint execution of different components: the program, the debugger, the monitor and the scenario. We formally describe the Interactively Runtime Verified program (i-RV-program) composed of these components as a Labeled Transition System (LTS). We first present each component

and our property model based on an extension of finite-state machines in Sec. IV-B. Events play the role of symbols of the LTS. Events are defined in Sec. IV-A. We then describe the evolution of the i-RV-program in Sec. IV-C using pseudo-code. This formalization is not needed to adopt the approach. However, it offers a programming-language independent basis for implementation and for reasoning over the concepts behind i-RV. In the extended version of this paper[4], we prove Theo. 1.

Theorem 1. The execution of the program is not affected by the presence of the monitor and the debugger and thus all executions observed through i-RV are faithful.

a) Notations: We define some notations used in this paper. We denote the set of booleans by $\mathbb{B} = \{\mathbf{T}\,(\text{true}), \mathbf{F}\,(\text{false})\}$ Given two sets E and F, $E \to F$ denotes the set of functions from E to F. By $f : E \to F$ or $f \in [E \to F]$, we denote that $f \in E \to F$. Let $f : E \longrightarrow F$, function $f' = f[x_1 \mapsto v]$ is such that $f'(x) = f(x)$ for any $x \neq x_1$, and $f'(x_1) = v$. The domain of function f is denoted by $\mathcal{D}(f)$.

Let us consider a non-empty set of elements E. The powerset of E is denoted $\mathcal{P}(E)$. Moreover, ϵ_E is the empty sequence (over E), noted ϵ when clear from the context. E^* denotes the set of finite sequences over E. Given two sequences s and s', the sequence obtained by concatenating s' to s is denoted $s \cdot s'$. We denote by Names the set of valid function and variable names in a program.

We define the transitive relation "f' *is more specific than* f": $f \sqsubseteq f' \stackrel{\text{def}}{=} \mathcal{D}(f) \subseteq \mathcal{D}(f') \wedge \forall p \in \mathcal{D}(f) : f(p) = f'(p)$. We also define the symetric relation "f' *is compatible with* f": $\mathrm{compat}(f, f') \stackrel{\text{def}}{=} \forall p \in \mathcal{D}(f) \cap \mathcal{D}(f') : f(p) = f'(p)$.

A. Events

i-RV is based on capturing events from the program execution with the debugger.

Definition 1 (Event)**.** An event is a tuple $e = (t, n, p, i, b) \in \mathrm{EventTypes} \times \mathrm{Names} \times \mathrm{Params}^* \times \mathrm{Values}_p^* \times \mathbb{B})$ where $\mathrm{EventTypes} = \{\texttt{Call}, \texttt{ValueWrite}, \texttt{ValueRead}, \texttt{UpdateExpr}\}$. The event name $n \in \mathrm{Names}$ is denoted $\mathrm{name}(e)$. Valid parameter names in Params are: v (a defined variable), $*p$ (the value pointed by p, with $p \in \mathrm{Params}$), $\&p$ (the address of variable p), $\arg i$ (the current value of parameter of index i) and ret (the "return value", which depends on the event type). If e is a symbolic event, its parameters are uninstantiated, i.e., $i = \emptyset$. If e is a runtime event, i is a list of parameter instances and $\mathrm{values}(e) \colon \mathrm{Names} \to \mathrm{Values}$ maps parameters to their values: $(\mathrm{values}(e))(p_k) = i_k$. Symbolic events are used to describe properties. Runtime event are *matched with* symbolic events if all its components, except values, are identical to the components of the symbolic event.

Example 1 (Event)**.** $(\mathrm{FunctionCall}, \texttt{push}, (q, v), \emptyset, \mathbf{T})$ is an event that is triggered before the call to function `push`. Parameters q and v are retrieved when producing the event.

The type $t \in \mathrm{EventTypes}$ of event e is denoted $\mathrm{type}(e)$. If $b = \mathbf{T}$ (resp. $\mathbf{F}$), e is a before (resp. after) event and $\mathrm{isBefore}(e) = \mathbf{T}$ (resp. $\mathbf{F}$). We describe the different event types. A $\mathrm{FunctionCall}$ event is generated by a function call. A before event is triggered before the first instruction of the function and after the jump to the function body. An after event fires after the last instruction of the function and before the jump to the caller. The parameter ret then corresponds to the return value of the call. A $\mathrm{ValueWrite}$ event is generated by an assignment. A before (resp. after) event fires before (resp. after) the assignment instruction and parameter ret refers to the old (resp. new) value of the variable. A $\mathrm{ValueRead}$ event is generated by a variable read. A before event fires before (resp. after) the instruction that reads the variable and parameter ret refers to the value of the variable. An $\mathrm{UpdateExpr}$ event is generated whenever the value of an expression is changed. A before (resp. after) event e is fired before (resp. after) the update. For a before (resp. after) $\mathrm{UpdateExpr}$ event, parameter ret refers to the old (resp. new) value of the expression.

Remark 1. In practice, $\mathrm{FunctionCall}$ events are captured using breakpoints and $\mathrm{ValueWrite}$, $\mathrm{ValueRead}$ and $\mathrm{UpdateExpr}$ events are captured using watchpoints. An $\mathrm{UpdateExpr}$ event requires as many watchpoints as variables in the expression. Current debuggers hide this requirement by allowing setting watchpoints on expressions.

B. Modeling the Components of i-RV

We model the components of i-RV and their behaviors by giving their configurations. Our execution model is a composition of these configurations.

1) The Program: For the sake of generality, we define a platform-independent and language-independent abstraction of a program that is loaded in memory, which allows us to apply the runtime techniques used in i-RV. The memory is abstracted as a function that maps addresses to values.

Definition 2 (Memory)**.** A memory m is a function in $\mathrm{Mem} = [\mathrm{Address} \to \mathrm{Values}]$. Some addresses correspond to variables of the program and are therefore linked to symbol names by the *symbol table* built during the compilation of the program.

Remark 2. The actual type of the elements of $\mathrm{Address}$ does not matter. They can be seen as integers like in a real memory. Elements of Values are machine words. They are either data (values of variables) or program instructions. They can also be seen as integers.

Definition 3 (Program)**.** A *program* is a 4-tuple $(\mathrm{Sym}, m_p^0, \mathrm{start}, \mathrm{runInstr})$ where $\mathrm{Sym} : \mathrm{Names} \to \mathrm{Address}$ is a *symbol table* $m_p^0 \in \mathrm{Mem}$ is the *initial memory*, $\mathrm{start} \in \mathrm{Address}$ is an address that points to the first instruction to run in the memory, and $\mathrm{runInstr} : (\mathrm{Mem} \times \mathrm{Address}) \to (\mathrm{Mem} \times \mathrm{Address} \times (\mathrm{Address} \times \mathbb{B} \times \mathbb{B})^*)$ is a function that abstracts the operational semantics of the program[1].

Function $\mathrm{runInstr}$ takes the current memory and Program Counter (PC) (in $\mathrm{Address}$) and executes the instruction at PC: it returns a (possibly new) memory, a new PC and a list of 3-tuples made of an address, and two booleans, representing the accesses to the memory. In an access, the two booleans hold true if the value at the given address was read and written (respectively), false otherwise. Memory accesses are used by the debugger to trigger watchpoints (see Sec. IV-B3).

[1]The actual semantics usually depends on the instruction set of the architecture.

Example 2 (Program)**.** In the remainder of this section, we will use program P given by the following source code to illustrate the concepts:

```
a := 0 ; b := 1 ; a := a + b
```

Definition 4 (Configuration of the program)**.** A *configuration* is a pair $(m_p, pc) \in \mathrm{Mem} \times \mathrm{Address}$ where m_p is the memory and pc is the current PC (an address in the program memory), i.e. the address of the next instruction.

Example 3 (Configuration of the program)**.** For program P given in Ex. 2, just after the execution of the second instruction, the configuration of the program is (m_p, pc_3) where pc_3 is the address of the code that corresponds to the third instruction of P, $m_p[\mathrm{Sym}(a)] = 0]$ and $m_p[\mathrm{Sym}(b)] = 1]$.

2) The Monitor: The monitor evaluates a property against a trace, giving a verdict upon the reception of each event. The verdict corresponding to the last event of the execution trace is called the final verdict [20].

a) Property model: We describe properties in a model based on finite-state machines. It is composed of states, transitions and an environment and it recognizes sequences of events. Transitions have guards that are expressions of event parameters and the memory and a function that can update the environment. Properties can be expressed on the whole set of events that can be retrieved from the debugger. Events are parameterized, i.e. values are linked to events. For instance, a function call generates an event parameterized with the values of arguments passed during this call, as well as values that are accessible at this time (global variables for example).

b) Trace slicing: Some properties should hold on each instance of an object or a set of objects rather than on the global state of the program. For example, a property on good file usage must be checked on each file that is manipulated by the program. For these properties, the execution trace is sliced in a way that is similar to what is achieved by trace slicing in [16], [21]. Each slice of the trace concerns a specific instance of an object or a set of objects on which the property holds. When trace slicing is used, a monitor does not correspond to a single finite state machine but to a set of finite state machines, one for each particular instance of an object.

Definition 5 (Monitor)**.** A *monitor* is a 7-tuple $(Q, \Sigma, \mathrm{init}, \mathrm{env}_0, \Delta, v, S)$ where Q is a set of states, Σ is the set of symbolic events, $\mathrm{env}_0 \in \mathrm{Env}$ is the initial environment ($\mathrm{Env} = \mathrm{Names} \to \mathrm{Values}$, where Names is the set of variable names and Values is the set of values that can be stored in a variable), $\Delta : \mathcal{P}(\mathrm{Names} \times \mathrm{Names}) \times Q \times \Sigma \times (\mathrm{Env} \times \mathrm{Env} \to \mathbb{B}) \times (\mathrm{Env} \times \mathrm{Env} \to \mathrm{Env}) \times Q$ is the transition relation, $v \in [Q \to \mathcal{V}]$ is the function mapping states to verdicts and $S \subseteq \mathrm{Names}$ is a set of slicing parameter names.

A *transition* is a 6-tuple $(sb, q_s, e_f, g, upd, q_d)$ where sb is the slice binding of the transition, q_s is the start state, e_f is the symbolic event, g is the guard, upd is the "updater" and q_d is the destination state. The slice binding sb is a set of pairs $(p_{\mathrm{prog}}, p_{\mathrm{prop}})$ where p_{prog} is the name of a parameter in the program and $_{\mathrm{prop}} \in S$ is the name of a slice parameter at the level of the property. The parameter p in the program is bound to the parameter s defined in the property. The guard $g : \mathrm{Env} \times \mathrm{Env} \to \mathbb{B}$ takes the environment built from the parameters of the runtime event, the environment of the monitor and returns a boolean. If it returns true (resp. false), the transition is taken (resp. not taken). The updater $upd : \mathrm{Env} \times \mathrm{Env} \to \mathrm{Env}$ returns an environment from the environment built from the parameters of the runtime event and the environment of the monitor. This function is used to update the environment of the property.

Example 4 (Monitor)**.** The property illustrated in Fig. 1 is a tuple $(Q, \Sigma, \mathrm{init}, \mathrm{env}_0, \Delta, v, S)$ where: $Q = \{\mathrm{Init}, \mathrm{ready}, \mathrm{sink}\}$, $\Sigma = \{e_f^{\texttt{before}}(\mathrm{queue\_new}), e_f^{\texttt{before}}(\mathrm{push}), e_f^{\texttt{before}}(\mathrm{pop})\}$, $\mathrm{init} = \mathrm{Init}$, $\mathrm{env}_0 = [N \mapsto 0, \mathrm{Max} \mapsto 0]$, $v = [\mathrm{Init} \mapsto \mathbf{T}, \mathrm{ready} = \mathbf{T}, \mathrm{sink} = \mathbf{F}]$, $S = \{q\}$, and the transition Δ is defined as

$\Delta = \{(\{(q,q)\}, \mathrm{Init}, e_f^{\texttt{before}}(\mathrm{new}), [any \mapsto \mathbf{T}],$
$([\mathrm{size}], \mathrm{env}) \mapsto \mathrm{env}[max := size - 1], \mathrm{ready}),$
$(\{(q,q)\}, \mathrm{ready}, e_f^{\texttt{before}}(\mathrm{push}), [[N, \mathrm{Max}] \mapsto N < \mathrm{Max}],$
$(any, \mathrm{env}) \mapsto \mathrm{env}[N+ = 1], \mathrm{ready}),$
$(\{(q,q)\}, \mathrm{ready}, e_f^{\texttt{before}}(\mathrm{pop}), [[N, \mathrm{Max}] \mapsto N > 0],$
$(any, \mathrm{env}) \mapsto \mathrm{env}[N- = 1], \mathrm{ready}),$
$(\{(q,q)\}, \mathrm{ready}, e_f^{\texttt{before}}(\mathrm{push}), [[N, \mathrm{Max}] \mapsto N >= \mathrm{Max}],$
$(any, \mathrm{env}) \mapsto \mathrm{env}, \mathrm{sink}),$
$(\{(q,q)\}, \mathrm{ready}, e_f^{\texttt{before}}(\mathrm{pop}), [[N, \mathrm{Max}] \mapsto N <= 0],$
$(any, \mathrm{env}) \mapsto \mathrm{env}, \mathrm{sink})\}$

The first transition makes the property transition from Init to ready when queue_new is called. The guard always returns true so the transition is taken unconditionally. The updater stores the maximum number of elements in the queue in the environment of the monitor. This maximum is computed from the size parameter of the event new. The two next transitions make the monitor stay on the state ready when it is correct to add or (resp. remove) elements from the queue. In each case, the updater updates the number of elements in the queue in the environment of the monitor. The two last transitions detect that an element is added (resp. removed) though the queue is full (resp. empty) and makes the property transition from ready to sink. Each time a new value of the parameter q is encountered, a new instance of the property is created.

Definition 6 (Configuration of the monitor)**.** A *configuration of the monitor* is a set of 4-tuples $M = \{(q_m^0, m_m^0, s_m^0, sp_m^0), \ldots (q_m^n, m_m^n, s_m^n, sp_m^n)\} \in \mathcal{P}(Q_m \times (\mathrm{Names} \to \mathrm{Values}) \times (S \to \mathrm{Values} \cup \{\not\}) \times (S \to \mathrm{Values} \cup \{\not\}))$ where $\not$ corresponds to an uninstantiated value.

In a configuration of a monitor, each 4-tuple $(q_m^k, m_m^k, s_m^k, sp_m^k) \in M$ represents an instance of the extended automaton that corresponds to a slice of the trace. q_m^k is its current state, m_m^k its current environment, s_m^k a mapping that gives which instance of the parameters this slice corresponds to (the slice instance) and sp_m^k the parent slice instance of this slice, that is, the slice instance of the slice sp from which this slice was created (because an event with parameters more specific than the parameter instance of sp happened). We denote by C_m the set of configurations of a monitor and by $\mathrm{enabled}(M)$ the set of symbolic events to which the monitor is "sensitive" in M: For all q in Q_m, $\mathrm{enabled}(q)$ can be determined statically: $\mathrm{enabled}(q) = \{e \in \mathrm{Events} \mid \exists (sb, g, upd, q_d) : (sb, q, e, g, upd, q_d) \in \Delta_m\}$. See Example 5 for an illustration of $\mathrm{enabled}(q)$. When a runtime event e_i is triggered, a transition $(q_s, e_f, g, \mathrm{upd}, q_d)$ is taken if the current state is q_s, e_i matches e_f and $g(e_i, m_p) = \mathbf{T}$, where m_m is the current environment. If so, the memory and the state of the property

are updated: $m'_m = \mathrm{upd}(e_i, m_p)$, where m'_m denotes the new environment and q_d becomes the new state.

Example 5. We denote by $e_f^{\texttt{before}}(\phi(\mathrm{params}))$ the symbolic before event $(\mathrm{FunctionCall}, \phi, \mathrm{params}, \mathrm{isBefore})$ corresponding to a call to function ϕ. For the property in Fig. 1, $\mathrm{enabled}(\mathrm{Init}) = \{e_f^{\texttt{before}}(\mathrm{queue\_new})\}$, $\mathrm{enabled}(\mathrm{ready}) = \{e_f^{\texttt{before}}(\mathrm{push}(q)), e_f^{\texttt{before}}(\mathrm{pop}(q))\}$, $\mathrm{enabled}(\mathrm{sink}) = \emptyset$.

3) The Debugger: The debugger provides primitives to instrument the program: breakpoints and watchpoints. It also provides a primitive to save the current state of the program and restore it: checkpoints. These primitives can also be used by the user during an interactive debugging session. A breakpoint stops the execution at a given address $a \in Address$ and a watchpoint when a given address containing data of interest is accessed (read, written, or both).

Definition 7 (Breakpoint)**.** A *breakpoint* is a 3-tuple $(addr, instr, isUserBP)$ where $addr \in \mathrm{Address}$ is the address of the breakpoint in the program memory, $instr \in \mathrm{Values}$ is the instruction to restore when the breakpoint is removed, and $isUserBP \in \mathbb{B}$ is a boolean that holds $\mathbf{T}$ if the breakpoint was set by the user, and $\mathbf{F}$ if it was set by the monitor. The set of breakpoints is defined by $\mathcal{B}\mathrm{p} = \mathrm{Address} \times \mathrm{Instr} \times \mathbb{B}$.

As we shall see in Sec. IV-C, when a breakpoint is reached, the execution is suspended and the debugger takes control over it. When a breakpoint is set, the debugger stores the instruction that is at the address of the breakpoint to be able to restore it when the breakpoint is removed or when the instruction is to be executed.

Example 6 (Breakpoint)**.** A breakpoint set by the user on the second instruction of the program given in Ex. 2 is $(pc_2, b := 1, true)$ where pc_2 is the memory address at which the second instruction is loaded. The second instruction is stored as the second component of the tuple and the third component indicates that this breakpoint is set by the user.

Definition 8 (Watchpoint)**.** A *watchpoint* is a 4-tuple $(addr, read, write, isUserWP) \in \mathcal{W}\mathrm{p}$ where $addr$ is the *address* of the watchpoint in the program memory, $read$ (resp. $write$) is a Boolean that holds true if this watchpoint should be triggered when the memory is *read* (resp. *written*), $isUserWP$ is a Boolean that holds $\mathbf{T}$ if the watchpoint was set by the user, and $\mathbf{F}$ if it was set by the monitor. The set of watchpoints is defined by $\mathcal{W}\mathrm{p} = \mathrm{Address} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}$.

Example 7 (Watchpoint)**.** A watchpoint set by the user on variable `b` in the program given in Ex. 2 is $(\&b, \mathbf{F}, \mathbf{T})$ where $\&b$ denotes the address of variable `b` in the program memory. This watchpoint is triggered whenever variable `b` is written (but not when it is only read).

a) Checkpoint: When debugging, it can be useful to save the state of the program (e.g., before the occurrence of a misbehavior to determine its cause or to try alternative executions). A checkpoint can be set by the user as well as by the scenario. There is not syntactical element in the definition of a checkpoint as it only depends on runtime elements. The states of the monitor and of the program are both saved, allowing coherent states after restoration.

Definition 9 (Checkpoint)**.** A *checkpoint* is a 2-tuple $(c_p, c_m) \in \mathcal{C}\mathrm{p}$ where c_p is a configuration of the program (as per Definition 3), and c_m is a configuration of the monitor (as per Definition 5). The set of all possible checkpoints is defined by $\mathcal{C}\mathrm{p} = (\mathrm{Mem} \times \mathrm{Address}) \times C_m$.

Example 8 (Checkpoint)**.** For the program given in Ex. 2, the checkpoint $(([a \mapsto 0, b \mapsto 1], pc_3), M)$, when the third instruction is about to be executed, is such that:

- $[a \mapsto 0, b \mapsto 1]$ is the program memory,
- pc_3 is the location of the third instruction in the memory,
- M is the monitor configuration when the checkpoint is set.

b) Configuration of the debugger: The debugger can be either interactive (I), waiting for the user to issue commands and execute them, or passive (P), with the program executing normally until a breakpoint or a watchpoint is triggered or the user interrupts the execution. The debugger keeps track of the current breakpoints, watchpoints and of user's checkpoints.

Definition 10 (Configuration of the debugger)**.** A *configuration of the debugger* is a 4-tuple $(q_d, \mathscr{B}, \mathscr{W}, \mathscr{C}) \in \{\mathrm{I}, \mathrm{P}\} \times \mathcal{P}(\mathcal{B}\mathrm{p}) \times \mathcal{P}(\mathcal{W}\mathrm{p}) \times \mathcal{C}\mathrm{p}^*$ where q_d is the current mode of the debugger, either I (interactive) or P (passive), $\mathscr{B}$ and $\mathscr{W}$ are the *sequences of breakpoints and watchpoints* handled by the debugger, $\mathscr{C}$ is the *sequence of checkpoints* set by the user.

Sequences are used for $\mathscr{C}$, $\mathscr{W}$ and $\mathscr{B}$ in order to allow the user manipulate checkpoints, watchpoints and breakpoints by their index.

4) The Scenario: The scenario reacts to monitor events by executing actions that update the state of the program, of the debugger and of the scenario itself. We define actions, then reactions, and finally the scenario itself. Actions are executed when monitor events are received according to the notion of scenario reactions.

Definition 11 (Scenario action)**.** The set of possible actions, Actions, is constructed like the set of statements in a classical programming language in which it is also possible to set and remove breakpoints, watchpoints and checkpoints and restore checkpoints.

Definition 12 (Scenario reaction)**.** A scenario reaction is a 3-tuple $(lt, q_m, a) \in \{\texttt{entering}, \texttt{leaving}\} \times Q_m \times \mathrm{Actions}$, where lt determines the "moment of the reaction", q_m is the state of the monitor to which the reaction is attached, and a is an action to be executed. The set of scenario reactions is denoted SR.

The scenario reaction (lt, q_m, a) is triggered when entering (resp. leaving) state q_m in the monitor when $lt = \texttt{entering}$ (resp. $lt = \texttt{leaving}$). When (lt, q_m, a) is triggered, action a is executed. A scenario is specified by giving a list of reactions and an environment (memory) m_s used by actions to store values. If a transition starting and leading to the same state is taken by the monitor, this state is both left and entered.

Definition 13 (Scenario)**.** A scenario is a pair $(m_s^0, S) \in (\mathrm{Names} \to \mathrm{Values}) \times \mathrm{SR}^*$ where m_s^0 is an initial environment and S a list of scenario reactions.

Remark 3. S is a list (and not a set) because if a state-update

in the monitor triggers more than one scenario reactions, these reactions are handled in order in S.

Example 9 (Scenario)**.** Assuming that `x` a monitor state, the following listing describes a scenario:

```
1 accesses := 0
2
3 on entering state x do
4 accesses := accesses + 1
5 if accesses = 2 then
6  [do something]
7 else
8  [do something else]
```

This listing defines the scenario $([\texttt{accesses} \mapsto 0], ((\texttt{entering}, \texttt{x}, a)))$ where action a increments variable `accesses` and its behavior depends on the value of variable `accesses`, the environment $[\texttt{accesses} \mapsto 0]$ is the initial memory of the scenario and $(\texttt{entering}, \texttt{x}, a)$ is the unique associated reaction.

C. Gathering the Components

In this section, we give the representation of the state of the i-RV-program at each execution step (i.e. its configuration). We then describe its evolution by means of pseudo-code, precisely explaining how it transitions from one configuration to another. The i-RV-program is depicted in Figure 2. Let $P = (\mathrm{Sym}, m_0^p, \mathrm{start}, \mathrm{runInstr})$ be a program, $M = (Q_m, q_m^0, m_m^0, \Sigma_m, \Delta_m)$ a monitor and $S = (m_s^0, S)$ a scenario. The i-RV-program, denoted by i-RV(P, M, S), is defined as the composition of P, M and S synchronized on events. We first define the configurations of the i-RV-program in Sec. IV-C1 and the evolution of its configurations in Sec. IV-C3 driven by the instrumentation functions of debugger defined in Sec. IV-C2.

1) Configuration of the Composition: We define the configurations of the i-RV-program.

Definition 14 (Configuration of the i-RV-program)**.** A *configuration of i-RV*(P, M, S) is a 4-tuple $c_{\text{i-RV}} = (c_{\mathrm{P}}, c_{\mathrm{dbg}}, c_{\mathrm{M}}, c_{\mathrm{S}}) \in (\mathrm{Mem} \times \mathrm{Address}) \times (\{\mathbf{I}, \mathbf{P}\} \times \mathcal{B}\mathrm{p}^* \times \mathcal{W}\mathrm{p}^* \times \mathcal{C}\mathrm{p}^*) \times C_m \times \mathrm{Mem}_s$. The initial configuration of i-RV(P, M, S) is $c_{\text{i-RV}}^0 = ((m_p^0, pc_0), (\mathbf{I}, \varepsilon, \varepsilon, \varepsilon), \{(\mathrm{init}, m_m^0, * \mapsto \not\checkmark, \varnothing)\}, m_s^0)$.

A configuration is composed of the initial program memory, the start address of the program as the PC, the debugger is interactive and does not manage any breakpoint, watchpoint or checkpoint, the monitor has one slice instance of the property that is in its initial state and in its initial environment and all parameters of the slice are uninstantiated and the memory of scenario is its initial memory.

2) Instrumentation Functions of the Debugger: Breakpoints and watchpoints are used to monitor function calls. We define the following functions: setBP (sets a breakpoint), unsetBP (finds a breakpoint by its address and removes it from the memories of the program and the debugger), setWP (sets a watchpoint) and unsetWP (unsets a watchpoint).

To set a breakpoint, we need to write a special instruction in the program memory. When this instruction is encountered during the execution, the execution is suspended and the debugger takes control over it. We also need to keep the word we replace in memory, so when the execution is resumed from this breakpoint or when the breakpoint is removed, the special breakpoint instruction is replaced by the stored instruction in the program memory.

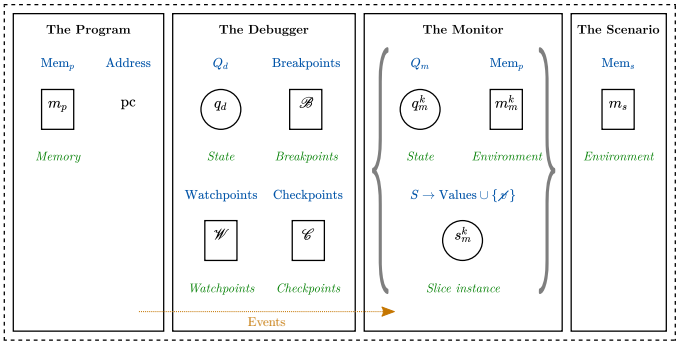

Figure 2. Configuration of a i-RV-program

Several breakpoints can be set at the same address. For instance, the monitor and the user might want to set a breakpoint on the same function. Breakpoints have to be stored in order in the structures of the debugger. We therefore use a list to save them.

We define setBP, a function that sets a breakpoint and register it in the configuration of the debugger. We indicate if the breakpoint was set by the user or by the monitor, so that breakpoints set by the user do not call the monitor and breakpoint set by the monitor are not seen by the user. We define $\mathrm{setInstrBp} : \mathrm{Env} \times \mathrm{Address} \to \mathrm{Env}$, a function replacing the word at a given address in the program memory by a breakpoint instruction (denoted by $\mathbf{B}$). $\mathrm{setInstrBp}(m_p, addr)[addr] = \mathbf{B}$ and $\forall a \in \mathrm{Address} : a \neq addr, \mathrm{setInstrBp}(m_p, a)[a] = m_k[a]$. Function $\mathrm{setBP} : \mathrm{Mem} \times \mathcal{B}\mathrm{p} \times \mathrm{Address} \times \mathbb{B}$ sets a breakpoint and saves it in the memory of the debugger: $\mathrm{setBP}(m_p, \mathscr{B}, addr, isUserBP) = (\mathscr{B}', m_p')$ with $m_p' = \mathrm{setInstrBp}(m_p, addr)$ and $\mathscr{B}' = (addr, m_p[addr], isUserBP) \cdot \mathscr{B}$. In the same way, we define $\mathrm{setWP} : \mathcal{W}\mathrm{p} \times \mathrm{Address} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}$ that adds a watchpoint in the memory of the debugger : $\mathrm{setWP}(\mathscr{W}, addr, r, w, isUserWP) = (addr, r, w, isUserWP) \cdot \mathscr{W}$. For watchpoints, the program memory does not need to be modified. We define unsetBP, which unsets a breakpoint and stops keeping track of it. $\mathrm{unsetBP}(m_p, \mathscr{B} \setminus \{bp\}, a, isUserBP) = (m_p', \mathscr{B}', b)$ s.t. bp is a breakpoint in $\mathscr{B}$ matching $(addr, \_, isUserBP)$ and $\forall addr \in \mathrm{Address}, m_p'[addr] = instr$ s.t. $(\_, instr, \_) = bp$ or $m_p'[addr] = m_p[a]$ otherwise. We also define $\mathrm{unsetWP}(\mathscr{W}, a, r, w, isUserWP) = (\mathscr{W} \setminus \{wp\}, wp)$ s.t. wp is a watchpoint in $\mathscr{W}$ matching $(addr, r, w, isUserWP)$.

Remark 4. Functions setInstrBp, setBP, unsetBP model the behavior of a real debugger using software breakpoints. A software breakpoint is implemented using a trap instruction. When several breakpoints are set at a single address, the debugger sets one trap instruction at this address in the memory of the program and keeps track of all the breakpoints in its internal structures.

For each monitor update, breakpoints and watchpoints corresponding to events monitored by the old (resp. new) state must be unset (resp. set). In Alg. 1, we define function INSTR used to set breakpoints needed for the new state. It takes the old program memory, the new state and the symbol table and returns a new memory and a new list of breakpoints. Function

REMOVEBPSWPS removes instrumentation that is not needed anymore.

Remark 5. Instrumenting to monitor value changes for an expression may require setting several watchpoints. The list of watchpoint to set for an expression is returned by function variablesAccesses : Events $\to \mathcal{P}(\mathcal{W}\text{p})$ which is not defined formally here for the sake of simplicity.

Algorithm 1 Instrumenting the Program

```
1: function NEEDSWATCHPOINT(e)
2:     return type(e) ∈ {ValueWrite ValueRead, UpdateExpr}
3: function EVTTOWPS(e, Sym)
4:     res ← ∅
5:     for all (n, r, w) ∈ variablesAccesses(e) do
6:         res ← res ∪ {(Sym[n], r, w)}
7:     return res
8: function REMOVEBPSWPS(B, W, M, Sym)
9:     for all e ∈ enabled(M) do
10:        if type(e) = FunctionCall then
11:            B' = B \ {b | b matches (addr, _, isUserBP)}
12:        if needsWatchpoint(e) then
13:            for all (a, r, w) ∈ EVTTOWPS(e, Sym) do
14:                W' ← unsetWP(W', a, r, w, F)
15:    return (B', W')
16: function INSTR(m_p, B, W, M, Sym)
17:    let (m'_p, B', W') ← (m_p, B, W)
18:    for all e ∈ enabled(M) do
19:        if type(e) = FunctionCall then
20:            (m'_p, B') ← setBP(m'_p, B', EVTTOBP(e, Sym), F)
21:        if needsWatchpoint(e) then
22:            for all (a, r, w) ∈ EVTTOWPS(e, Sym) do
23:                W' ← SETWP(W', a, r, w, F)
24:    return (m'_p, B', W')
```

In a checkpoint, the saved program memory must not contain any instruction $\mathbf{B}$. A function to remove all breakpoints from the memory is therefore needed. We define removeBPInstrs, a function which iterates over the list of breakpoints of the debugger and replaces each instruction $\mathbf{B}$ by the original instruction. $\text{removeBPInstrs}(m_p, \epsilon) = m_p$ and $\forall \mathscr{B} \in \mathcal{B}\text{p} : \mathscr{B} \neq \epsilon, \text{removeBPInstrs}(m_p, \mathscr{B}) = \text{removeBPInstrs}(m'_p, \mathscr{B}')$ where $\mathscr{B} = (addr, instr, b) \cdot \mathscr{B}'$ and $m'_p = m_p[addr \mapsto instr]$. When a checkpoint is restored, current breakpoints must be set in the memory. We therefore define the function restoreBPs which iterates over the list of breakpoints and sets the instruction $\mathbf{B}$ at the relevant addresses. $\text{restoreBPs}(m_p, \mathscr{B}) = m'_p$ with $\forall a \in \text{Address}, m'_p[a] = \mathbf{B}$ if $\exists (addr, instr, b) \in \mathscr{B} : addr = a$, $m_p[a]$ otherwise.

3) Evolution of the i-RV-program: In this section, we describe the precise behavior of the i-RV-program. The algorithm describing the general behavior of the i-RV-program is given in Alg. 2 and explained right after. The initial configuration of the i-RV-program is $((m_p, pc), (q_d, \mathscr{B}, \mathscr{W}, \mathscr{C}), M, m_s) = ((m^0_p, pc_0), (\text{I}, \varepsilon, \varepsilon, \varepsilon), \{(\text{init}, m^0_m, * \mapsto \not\downarrow, \varnothing)\}, m^0_s)$. In this configuration, the debugger is in interactive mode, meaning it is waiting for commands from the user (Line 3)

a) First step of the execution: When starting the execution of the i-RV-program, the monitor is initialized (command load monitor, Alg. 3, Line 4): breakpoints and watchpoints relevant to the initial state of the property are set by function *instr* that populates the lists $\mathcal{W}\text{p}$ and $\mathcal{B}\text{p}$ and alters the instructions of the program accordingly.

Algorithm 2 Behavior of the System

```
1: let cont ← T
2: while cont do
3:     if q_d = I then
4:         (cont, c_i-RV) ← HANDLEUSERCMD(c_i-RV)
5:     else if User stops the execution or m_p[pc] = stop then
6:         q_d ← I
7:     else if m_p[pc] ≠ B ∧ m_p[pc] ≠ stop then
8:         c_i-RV ← NORMALSTEP(c_i-RV)
9:     else if m_p[pc] = B then
10:        c_i-RV ← HANDLEBP(c_i-RV)
```

Algorithm 3 Behavior When the Debugger is Interactive

```
1: function HANDLEUSERCMD(c_i-RV)
2:     let cont ← T
3:     switch getUserCMD() do
4:         case load monitor
5:             (m_p, B, W) ← INSTR(m_p, B, W, M, Sym)
6:         case restart n
7:             (B, W) ← REMOVEBPSWPS(B, W, M, Sym)
8:             (m_p^tmp, pc, M) ← C_n
9:             (m_p, B, W) ← INSTR(restoreBPs(m_p^tmp, B),
10:                                B, W, M, Sym)
11:        case continue
12:            c_i-RV ← INTERACTIVESTEP(c_i-RV) ; q_d ← P
13:        case break a
14:            (m_p, B) ← if a ∈ Address: setBP(m_p, a, B, T)
15:                       else setBP(m_p, Sym(name), B, T)
16:        case watch mode a, a ∈ Address
17:            W ← W · (a, r ∈ mode, w ∈ mode, T)
18:        case checkpoint
19:            C ← C · (removeBPInstrs(m_p, B), M, pc)
20:        case step: c_i-RV ← INTERACTIVESTEP(c_i-RV)
21:        case exit: cont ← F
22:    return (cont, c_i-RV)
```

b) Normal execution: If the debugger is passive and the instruction about to be executed is not an instruction $\mathbf{B}$), the program executes normally as if there were no debugger and no monitor (Alg. 2), Line 7. In function NORMALSTEP (Alg. 7, Line 16), the PC and the program memory are updated by function runInstr which runs the instruction to be executed. Watchpoints relevant to memory accesses made by this execution are handled. The instruction **stop** ends the execution (Alg. 2, Line 5).

c) Handling a watchpoint: In Alg. 4, we define HANDLEWP. In the case of a user watchpoint, the state of the i-RV-program is returned as is, except for the state of the debugger, which becomes interactive. In the case of a monitor watchpoint, the corresponding events are applied using the function APPLYEVTS defined in Alg. 5, updating the monitorand executing the scenario.

d) The user sets a breakpoint: When the debugger is interactive (I), the user can set a breakpoint (Alg. 3, Line 13) by giving either an address in the program memory or a symbol (function) name transformed into an address using the symbol table Sym, part of the definition of the program. If the user issues a command to set a breakpoint at address a, the function setBP updates the current program memory m_p and the list of breakpoints $\mathscr{B}$ of the debugger. The resulting memory m'_p and list of breakpoints $\mathscr{B}'$ are stored in the configuration of the i-RV-program.

Algorithm 4 Handling Instrumentation (Generating Events)

1: **function** HANDLEBP($c_{\text{i-RV}}$)
2: | **if** $\exists\, instr : (addr, instr, \mathbf{T}) \in \mathscr{B}$ **then**
3: | | **return** $(m_p, pc), (\text{I}, \mathscr{B}, \mathscr{W}, \mathscr{C}), M, m_s)$
4: | **return** APPLYEVTS(bpToEvts(m'_p, pc, M, Sym), $c_{\text{i-RV}}$)
5: **function** WATCHPOINTSMATCHING($\mathscr{W}, (addr, r, w)$)
6: | $Wp_s \leftarrow \emptyset$
7: | **for all** $(addr', r', w', isUserWP) \in \mathscr{W}$ **do**
8: | | **if** $addr = addr' \wedge (r = r' \vee w = w')$ **then**
9: | | | $Wp_s \leftarrow \mathscr{W}_s \cup \{(addr', r', w', isUserWP)\}$
10: | **return** Wp_s
11: **function** HANDLEWP(accesses, $c_{\text{i-RV}}$)
12: | $\mathscr{W}_s \leftarrow \emptyset$
13: | **for all** access $\in$ accesses **do**
14: | | $\mathscr{W}_s \leftarrow \mathscr{W}_s \cup \{\text{watchpointsMatching}(\mathscr{W}, \text{access})\}$
15: | **if** $\exists(\_, \_, \_, \mathbf{T}) \in \mathscr{W}_s$ **then**
16: | | **return** $((pc, m_p), (\text{I}, \mathscr{B}, \mathscr{W}, \mathscr{C}), M, m_s)$
17: | **return** APPLYEVTS(wpsToEvts(Wp_s, P, M, Sym), $c_{\text{i-RV}}$)

e) The user sets a watchpoint: In interactive mode (I), the user can set a watchpoint by giving the address in the program memory where it should be set (Alg. 3, Line 16).

f) The user sets a checkpoint: In interactive mode, the user can set a checkpoint (Alg. 3, Line 18). Several objects are saved: the program memory (without the breakpoints instructions), the PC and the state of the monitor. The new checkpoint is appended to $\mathscr{C}$.

g) The user restarts a checkpoint: In interactive mode (I), the user can restore a checkpoint (Alg. 3, Line 6). The current program memory, PC and configuration of the monitor with its memory are restored from the checkpoint. Current breakpoints are set in the newly restored program memory (matching the behavior of GDB and LLDB).

h) A breakpoint instruction is encountered: When encountering a breakpoint instruction, the debugger has to check if it matches a breakpoint of the user or a breakpoint of the monitor. In the first case, the i-RV-program transitions to the I state. In the second case, the event is applied.

Remark 6. In real systems, the breakpoint instruction triggers a trap caught by the operating system which suspends the execution and informs the debugger of the trap. Traps are not described in our model because we do not model the OS. The behavior of our model is otherwise close to the reality.

i) Handling a breakpoint: In Alg. 4, we define HANDLEBP. If the breakpoint belongs to the user, the i-RV-program becomes interactive but is not otherwise modified. If the breakpoint belongs to the monitor, breakpoints are removed from the program memory, a corresponding before event is applied using the function APPLYEVENT defined in Alg. 5, the original instruction is run and an after event is applied using the function APPLYEVENT that updates the state of the i-RV-program. It first updates each slice of the configuration of the monitor according to the event and the transition relation, retrieving the set of transitions involved. It then applies the scenario using function APPLYSCENARIO defined in Alg. 6. The scenario can update the whole state of the i-RV-program. It is applied only if the current state has been updated (Line 18 of Alg. 5). For each entry of the scenario, if the event corresponds to the entry, the corresponding action is run with function runAction. For the sake of conciseness, function runAction is not defined. See Alg. 2, Line 9. Breakpoints are restored and the instrumentation needed for the new current state is added.

Algorithm 5 Handling Events

1: **function** UPDATEMON(M, e):
2: | $v \leftarrow \text{values}(e)$; $M' \leftarrow \emptyset$; $slicesUpd \leftarrow \emptyset$
3: | **for all** $(q, m, s, sp) \in M$ **do**
4: | | **for all** $(sb, q_s, e_t, g, upd, q_d) \in \Delta_m$ **do**
5: | | | $inst \leftarrow [p_{\text{prop}} \mapsto v(p_{\text{prog}}) \mid \exists\, (p_{\text{prog}}, p_{\text{prop}}) \in sb]$
6: | | | **if** $q = q_s \wedge e$ matches $e_t \wedge \text{compat}(s, inst)$ **then**
7: | | | | **if** $inst \sqsubseteq s$ **then**
8: | | | | | $M' \leftarrow M' \cup \{(q_d, upd(v, m), s, sp)\}$
9: | | | | | $slicesUpd \leftarrow slicesUpd \cup \{(q_s, q_d, s)\}$
10: | | | | **else if** $\nexists (q', m', s', sp') \in M : inst \sqsubseteq s'$ **then**
11: | | | | | $M' \leftarrow M' \cup \{(q_d, upd(v, m), inst, s)\}$
12: | | | | | $slicesUpd \leftarrow slicesUpd \cup \{(q_s, q_d, inst)\}$
13: | | | **else**
14: | | | | $M' \leftarrow M' \cup \{(q, m, s, sp)\}$
15: | **return** $(M', slicesUpd)$
16: **function** APPLYEVENT($c_{\text{i-RV}}, e$)
17: | $(M', slicesUpd) \leftarrow$ UPDATEMON(M, e)
18: | **return** APPLYSCENARIO($S, c_{\text{i-RV}}, slicesUpd$)
19: **function** APPLYEVTS($evList, c_{\text{i-RV}}$)
20: | $m'_p \leftarrow \text{removeBPInstrs}(m_p, \mathscr{B})$
21: | $(\mathscr{B}', Wp') \leftarrow$ REMOVEBPSWPS($\mathscr{B}, \mathscr{W}, M, \text{Sym}$)
22: | $(pc', q'_d, \mathscr{C}', M', m'_s) \leftarrow (pc, q_d, \mathscr{C}, M, m_s)$
23: | **for all** $e \in evList$ s.t. isBefore(e) **do**
24: | | $c'_{\text{i-RV}} \leftarrow$ APPLYEVENT($c'_{\text{i-RV}}, e$)
25: | $(m'_p, pc') \leftarrow \text{runInstr}(m'_p, pc')$
26: | **for all** $e \in evList$ s.t. not isBefore(e) **do**
27: | | $c_{\text{i-RV}} \leftarrow$ APPLYEVENT($c_{\text{i-RV}}, e$)
28: | $m'^{tmp}_p \leftarrow \text{restoreBPs}(m'_p, \mathscr{B}')$
29: | $(m'_p, \mathscr{B}', \mathscr{W}') \leftarrow$ INSTR($m'^{tmp}_p, \mathscr{B}', \mathscr{W}', M', \text{Sym}$)
30: | **return** $c'_{\text{i-RV}}$

Algorithm 6 Applying the Scenario

1: **function** SRMATCHESEVT($lt, q_{\text{scn}}, q_s, q_d$)
2: | **return** $(lt = \texttt{leaving} \wedge q_{\text{scn}} = q_s) \vee (lt = \texttt{entering} \wedge q_{\text{scn}} = q_d)$
3: **function** APPLYSCENARIO($scenario, c_{\text{i-RV}}, slicesUpd$)
4: | **if** $scenario = \epsilon$ **return** $c_{\text{i-RV}}$
5: | $(lt, q_{\text{scn}}, a) \leftarrow \text{head}(scenario)$
6: | **for all** $(q_s, q_d, inst) \in slicesUpd$ **do**
7: | | **if** SRMATCHESEVT($lt, q_{\text{scn}}, q_s, q_d$) **then**
8: | | | $c_{\text{i-RV}} \leftarrow \text{runAction}(c_{\text{i-RV}}, inst, a)$
9: | **return** APPLYSCENARIO(tail($scenario$), $c_{\text{i-RV}}, slicesUpd$)

j) The execution of the program is done step by step: In interactive mode, function interactiveStep (Alg. 7) is executed when the command `step` is issued. The instruction at the current address is run normally and possible watchpoints are handled (Lines 20 and 21). If the instruction is a breakpoint, the breakpoint is handled if it is set by the monitor or ignored otherwise, and the original instruction is executed.

k) The execution of the program is interrupted by the user: The debugger switches from passive to interactive mode (Line 5 of Alg. 2). This is meaningful if a step in the algorithm is assumed to take a non-zero amount of time.

l) The execution of the program is resumed: If the execution is continued (e.g. by issuing the command `continue`, see Line 11), a step is executed (in case the execution was interrupted by a breakpoint or a watchpoint) and the i-RV-program transitions from I to P.

Algorithm 7 Handling a Step

function HANDLESTEPWP(accesses, $(m_p^{\text{tmp}}, pc^{\text{tmp}}), c_{\text{i-RV}}$))
  $\mathscr{W}_s \leftarrow \emptyset$
  for all access $\in$ accesses **do**
    $\mathscr{W}_s \leftarrow \mathscr{W}_s \cup \{\text{watchpointsMatching}(\mathscr{W}, \text{access})\}$
  $\mathscr{W}_s \leftarrow \{(addr, r, w, isUserWP) \in \mathscr{W}_s \mid isUserWP = \mathbf{F}\}$
  if $\mathscr{W}_s = \emptyset$ **return** $(m_p^{\text{tmp}}, pc^{\text{tmp}}), (\mathbf{P}, \mathscr{B}, \mathscr{C}), M, m_s)$
  return APPLYEVTS(wpsToEvts($\mathscr{W}_s, m_p, pc, M, \text{Sym}$), $c_{\text{i-RV}}$)
function HANDLESTEPBP($((m_p, pc), D, M, m_s)$)
  if $\exists instr : (pc, instr, \mathbf{F}) \in \mathscr{B}$ **then**
    return APPLYEVTS(bpToEvts(m'_p, pc, M, Sym), $c_{\text{i-RV}}$)
  let $instr : (pc, instr, \_) \in \mathscr{B}$ such that $instr \neq \mathbf{B}$
  $c'_{\text{i-RV}} \leftarrow$ NORMALSTEP($c_{\text{i-RV}}$)
  if $\exists instr : (pc, instr, \mathbf{T}) \in \mathscr{B}'$ **then**
    return $((m'_p[pc \mapsto \mathbf{B}], pc'), D', M', m'_s)$
  return $((m'_p, pc'), (q'_d, \mathscr{B}', \mathscr{W}'), M', m'_s)$
function NORMALSTEP($c_{\text{i-RV}}$)
  $(m_p^{\text{tmp}}, pc^{\text{tmp}}, \text{accesses}) \leftarrow \text{runInstr}(m_p, pc)$
  return HANDLEWP(accesses, $c_{\text{i-RV}}$)
function INTERACTIVESTEP($c_{\text{i-RV}}$)
  if $m_p[pc] = \mathbf{B}$ **return** HANDLESTEPBP($c_{\text{i-RV}}$)
  if $m_p[pc] \neq$ **stop then**
    return NORMALSTEP($(m_p, pc), (\mathbf{P}, \mathscr{B}, \mathscr{C}), M, m_s$)
  print "Illegal Command"

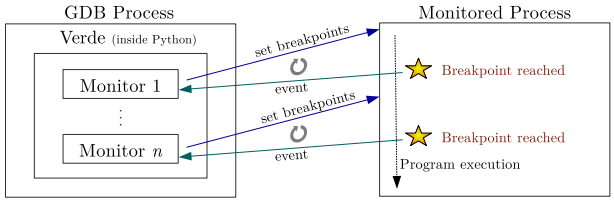

Figure 3. Instrumentation in Verde

V. Implementation: Verde

To evaluate our approach in terms of usefulness and performance, we implemented it in a tool called Verde. We overview Verdeand give some details about its architecture in Sec. V-A. In Sec. V-B, we describe the syntax used in Verde to write properties. We explain how to use Verde in Sec. V-D.

A. Overview

Verde[2] is written in Python and works seamlessly as a GDB plugin by extending GDB Python interface. Verde can be used with any program written in a programming language supported by GDB. Verde supports the verification of several properties by means of monitors working independently. Each monitor sets and deletes breakpoints according to the events that are relevant to its current state. Verde provides a graphical and animated view of the properties being checked at runtime.

[2]Verde can be downloaded at https://gitlab.inria.fr/monitoring/verde.

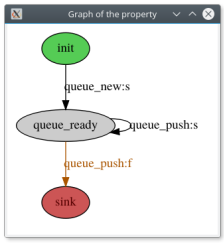

Figure 4. Graphical view the property given in Figure 5.

```
slice on queue
initialization: {N = 0; maxSize = 0}
state init accepting:
    transition:
        event queue_new(queue, size : int) { maxSize = size − 1 }
        success queue_ready
state queue_ready accepting:
    transition:
        event queue_push(queue, prod_id) { return N < maxSize }
        success { N = N + 1; print("nb elem: "+str(N));} queue_ready
        failure { print("%d made %d overflow!"% (prod_id, queue)) } sink
    transition:
        event queue_pop(queue, prod_id) { return N > 0 }
        success { N = N − 1; print("nb elem: "+str(N)) } queue_ready
        failure sink
state sink non−accepting sink_reached()
```

Figure 5. Verde version of the property in Figure 1

The view facilitates understanding the current evaluation of the property and, as a consequence, the program. Verde also lets the developer control the monitors and access their internal state (property instances, current states, environments). Verde is called by GDB when GDB handles breakpoints in the monitored program that were set by Verde. When a breakpoint is hit, the state of the property is updated and the execution is resumed. Fig. 3 depicts the execution of a program with Verde.

B. Syntax of Properties

Verde provides a DSL for writing properties in the model presented in Sec. IV-B2 with slight modifications to allow more conciseness[3].Fig. 5 gives a property used to check whether an overflow happens in a multi-threaded producer-consumer program. First, the optional keyword `slice on` gives the list of slicing parameters. Then, an optional Python code block initializes the environment of the monitor. Then, states are listed, including the mandatory state `init`. A state has a name, an optional annotation indicating whether it is accepting, an optional action name attached to the state and its transitions. Transitions can be written with two destination states: a success (resp. failure) state used when the guard returns SUCCESS (resp. FAILURE). The transition is ignored if the guard returns NOT RELEVANT. Each transition comprises the monitored event, the parameters of the event used in the guard, the guard (optional), the success block and the failure block (optional). Success and failure blocks comprise an optional Python code block, an optional action name and the name of a destination state. The guard is a side-effect free Python code block that returns `True` (resp. `False`) if the guard succeeds (resp. fails) and None if the transition should be ignored.

C. Checkpointing

Verde features two process checkpointing techniques on Linux-based systems. The first uses the native `checkpoint` command of GDB. This method is based on `fork()` to save the program state in a new process, which is efficient, as `fork` is implemented using Copy on Write. A major drawback of this technique is that multithreaded programming is not supported since `fork()` keeps only one thread in the new process. The second technique uses CRIU[4], which supports multithreaded

[3]We did not use pre-existing syntax in order to allow us flexibility as we experiment. Interfacing with existing monitoring tool is planned.

[4]Checkpoint/Restore In Userspace

processes and trees of processes. CRIU uses the `ptrace` API to attach the (tree of) process to be checkpointed and saves its state in a set of files. CRIU supports incremental checkpointing by computing a differential between an existing checkpoint and a checkpoint to create. It can make the system track memory changes in the process to speed this computation.

D. Using Verde

A typical usage session begins by launching `gdb` and Verde (which can be automatically loaded by configuring GDB appropriately). Then, the user loads one or several properties. Additional python functions, used in properties, can be loaded at the same time. A scenario can also be loaded. Then, the user starts the execution. It is also possible to display the graph of the property with the `show-graph` subcommand (see Fig. 4).

```
$ gdb ./my-application
(gdb) verde load-property behavior.prop functions.py
(gdb) verde load-scenario default-scenario.sc
(gdb) verde show-graph
(gdb) verde run-with-program
...
[verde] Initialization: N = 0
[verde] Current state: init (N = 0)
queue.c: push!
[verde] Current state: init
...
queue.c: push!
[verde] GUARD: nb push: 63
[verde] Overflow detected!
[verde] Current state: sink (N = 63)
[Execution stopped.]
(gdb)
```

VI. Evaluation

We report on six experiments carried out with Verde to measure its usefulness in finding and correcting bugs and its efficiency from a performance point of view[5]. We discuss the objective and possible limitations (threat to validity) of each experiment. These experiments also illustrate how a developer uses Verde in practice.

A. Correcting a Bug in `zsh`

In `zsh`, a widely-used UNIX shell, a segmentation fault happened when trying to auto-complete some inputs like `!>.` by hitting the tab key right after character `>`.

We ran `zsh` in GDB, triggered the bug and displayed a backtrace leading to a long and complicated function, `get_comp_string`, calling another function with a null parameter `itype_end`, and then making `zsh` crash. Instead of trying to read and understand the code or debugging step by step, we observed the bug (null pointer) and inspected the stack trace. We noticed a call to function `itype_end` with a null parameter. Then, we wrote a property tracking assignments related to this variable and checking that this variable, whenever used, is not null, and a scenario that prints the backtrace each time the state of the property changes. This let us see that the last write to this variable nulls it. We were able to prevent the crash by adding a null check before a piece of code that seems to assume that the variable is not null and that contains the call to `itype_end` leading to the crash[6]. We did not discover the bug using i-RV[7]. However, it helped us determine its origin in the code of `zsh` and fix it. A fix has since been released.

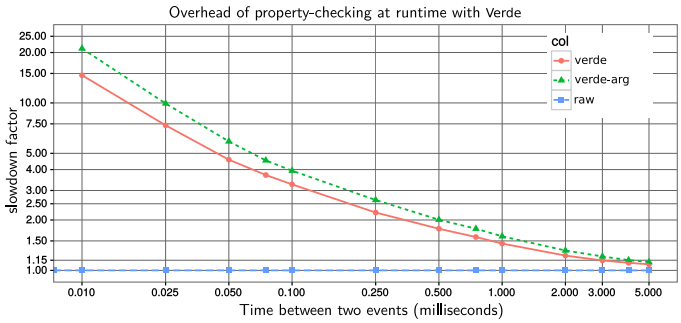

Figure 6. Instrumentation overhead with Verde.

B. Multi-Threaded Producer-Consumers

This experiment is purposed to check whether our approach is realistic in terms of usability. We considered the following use-case: a developer works on a multi-threaded application in which a queue is filled by 5 threads and emptied by 20 threads and a segmentation fault happens in several cases. We wrote a program deliberately introducing a synchronization error, as well as a property (see Fig. 1) on the number of additions in a queue in order to detect an overflow. The size of the queue is a parameter of the event `queue_new`. The function `push` adds an element into the queue. A call to this function is awaited by the transition defined at line 15 of Fig. 5. We ran the program with Verde. The execution stopped in the state `sink` (defined at line 39 of Fig. 5). In the debugger, we had access to the precise line in the source code from which the function is called, as well as the complete call stack. Under certain conditions (that we artificially triggered), a mutex was not locked, resulting in a queue overflow. After fixing this, the program behaved properly. In this experiment, we intentionally introduced a bug (and thus already knew its location). However this experiments validates the usefulness of Verde in helping the programmer locate the bug: the moment the verdict given by the monitor becomes false can correspond to the exact place the error is located in the code of the misbehaving program.

C. Micro-benchmark

In this experiment, we evaluated the overhead of the instrumentation in function of the temporal gap between events. We wrote a C program calling a NOP function in a loop. To measure the minimal gap between two monitored events for which the overhead is acceptable, we simulated this gap by a loop of a configurable duration. The results of this benchmark using a Core i7-3770 @ 3.40 GHz (with a quantum time (process time slice) around 20 ms), under Ubuntu 14.04 and Linux 3.13.0, are presented in Fig. 6. The curve `verde-arg` corresponds to the evaluation of a property which retrieves an argument from calls to the monitored function. With 0.5 ms between two events, we measured a slowdown factor of 2. Under 0.5 ms, the overhead can be significant. From 3 ms, the slowdown is under 20 % and from 10 ms, the slowdown is under 5 %. We noticed

[5] A video and the source codes needed for reproducing the benchmarks are available at http://gitlab.inria.fr/monitoring/verde.

[6] We worked on commit 85ba685 of `zsh`.

[7] The bug was reported at https://sourceforge.net/p/zsh/bugs/87/

that the overhead is dominated by breakpoint hits. The absolute overhead by monitored event, in the manner of the overhead of an argument retrieval, is constant. We measured the mean cost of encountering a breakpoint during the execution. We obtained 95 µs on the same machine and around 300 µs on a slower machine (i3-4030U CPU @ 1.90 GHz). While this experiment does not give a realistic measure of the overhead added by the instrumentation, it is still useful to estimate the overhead in more realistic scenarios.

D. User-Perceived Performance Impact

a) Multimedia Players and Video Games: We evaluated our approach on widespread multimedia applications: the VLC and MPlayer video players and the SuperTux 2D platform video game. A property made the monitor set a breakpoint on the function that draws each frame to the screen for these applications, respectively *ThreadDisplayPicture*, *update_video* and *DrawingContext::do_drawing*. For SuperTux, the function was called around 60 times per second. For the video players, it was called 24 times per second. In each case, the number of frames per second was not affected and the CPU usage remains moderated: we got an overhead of less than 10 % for the GDB process. These results correspond to our measurements in Sec. VI-C: there is a gap of 16 ms between two function calls which is executed 60 times per second. Thus, our approach does not lead to a significant overhead for multimedia applications when the events occur at fixed frequency.

b) Opening and Closing Files, Iterators: We evaluated the user-perceived overhead with widespread applications. We ensured that all open files are closed with the Dolphin file manager, the NetSurf Web browser, the Kate text editor and the Gimp image editor. Despite some slowdowns, caused by frequent disk accesses, they remained usable. Likewise, we checked that no iterator over hash tables of the GLib library (`GHashTableIter`) that is invalidated was used. Simplest applications like the Gnome calculator remained usable but strong slowdowns were observed during the evaluation of this property, even for mere mouse movements. In Sec. VII, we present possible ways to mitigate these limitations.

E. Automatic Checkpointing to Debug a Sudoku Solver

We evaluated i-RV by mutating the code of a backtracking Sudoku solver[8]. This experiment illustrates the use of scenarios to automatically set checkpoints and add instrumentation at relevant points of the execution. Sudoku is a game where the player fills a 9x9 board such that each row, each column and each 3x3 box contains every number between 1 and 9. The solver reads a board with some already filled cells and prints the resulting board. During the execution, several instances of the board are created and unsolvable instances are discarded. We wrote a property describing its expected global behavior after skimming the structure of the code, ignoring its internal details. No values should be written on a board deemed unsolvable or that break the rules of Sudoku (putting two same numbers in a row, a column or a box). Loading a valid board should succeed. We then wrote a scenario that creates checkpoints whenever the property enters an accepting state. Entering a non-accepting state makes the scenario restore the last checkpoint and add watchpoints on each cells of the concerned board instance. When watchpoints are reached, checkpoints are set, allowing us to get a more fine-grained view of the execution close to the misbehavior and choose the moment of the execution we want to debug. This scenario allows a first execution that is not slowed down by heavy instrumentation, and precise instrumentation for a relevant part of it. The solver is bundled with several example boards that it solves correctly. We mutated its code using `mutate.py`[9] to artificially introduce a bug without us knowing where the change is. When ran, the mutated program outputs "bad board". We ran it with i-RV. The property enters the state `failure_load`. When restoring a checkpoint and running the code step by step in the function that loads a board, the execution seems correct. The code first runs one loop reading the board using `scanf` by chunks of 9 cells, and then a second loop iterates over the 81 cells to convert them to the representation used by the solver. Setting breakpoints and displaying values during the first loop exhibits a seemingly correct behavior. During the second loop, the last line of the board holds incorrect values. Since we observed correct behavior for the first loop and the 72 first iterations of the second loop, and since both loops do not access the board in the same way, we suspected a problem with the array containing the board. We checked the code and saw that the mutation happened in the type definition of the board, giving it 10 cells by line instead of 9. A caveat of this experiment is that we had to choose the mutated version of the code such that the code violates the property. We also introduced a bug artificially rather than working on a bug produced by a human. However, the example can be generalized and illustrates how scenarios can be used for other programs, where checkpoints are set on a regular basis and execution is restarted from the last one and heavy instrumentation like watchpoints is used, restricting slowness to a small part of the execution.

VII. Future Work

a) Instrumentation: Handling breakpoints is costly [22] and handling watchpoints even more. Code injection could provide better efficiency [23], [24] by limiting round trips between the debugger and the program would to the bare minimum while keeping the current flexibility of the approach.

b) Checkpointing the File System: We plan to explore the possibility of capturing the environment of the developer in addition to the process being debugged when checkpointing. More specifically, we shall look at the atomic snapshotting capabilities of modern file systems like Brfs and ZFS.

c) Record and Replay and Reverse Execution: RR is a powerful technique for finding bugs. Once a buggy execution is recorded, the bug can be studied and observed again by running the recording. We aim to augment i-RV with reverse debugging techniques.

d) Usability and Scalability: Our biggest experiment involves a medium-sized sized application, Zsh (4 MiB of source code), and has been conducted ourselves. The next step is to show that it indeed eases bug fixing with bigger applications and conduct a solid user study.

[8] https://github.com/jakub-m/sudoku-solver

[9] https://github.com/arun-babu/mutate.py

References

[1] K. Havelund and A. Goldberg, "Verify your runs," in *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, ser. Lecture Notes in Computer Science, B. Meyer and J. Woodcock, Eds., vol. 4171. Springer, 2005, pp. 374–383. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69149-5_40

[2] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jlap.2008.08.004

[3] O. Sokolsky, K. Havelund, and I. Lee, "Introduction to the special section on runtime verification," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 243–247, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10009-011-0218-6

[4] R. Jakse, Y. Falcone, J. Méhaut, and K. Pouget, "Interactive runtime verification," *CoRR*, vol. abs/1705.05315, 2017. [Online]. Available: http://arxiv.org/abs/1705.05315

[5] R. Jakse, Y. Falcone, J.-F. Méhaut, and K. Pouget, "Verde repository," 2017, https://gitlab.inria.fr/monitoring/verde.

[6] J. Engblom, "A review of reverse debugging," in *System, Software, SoC and Silicon Debug Conference (S4D), 2012*. IEEE, 2012, pp. 1–6.

[7] K. Georgiev and V. Marangozova-Martin, "Mpsoc zoom debugging: A deterministic record-partial replay approach," in *12th IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2014, Milano, Italy, August 26-28, 2014*, 2014, pp. 73–80. [Online]. Available: http://dx.doi.org/10.1109/EUC.2014.20

[8] J. Roos, L. Courtrai, and J. Méhaut, "Execution replay of parallel programs," in *1993 Euromicro Workshop on Parallel and Distributed Processing, PDP 1993, Gran Canaria, Spain, 27-29 January 1993*, 1993, pp. 429–434. [Online]. Available: http://dx.doi.org/10.1109/EMPDP.1993.336375

[9] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[10] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06. New York, NY, USA: ACM, 2006, pp. 73–85. [Online]. Available: http://doi.acm.org/10.1145/1217935.1217943

[11] T. Ball and S. K. Rajamani, "The slam project: Debugging system software via static analysis," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02. New York, NY, USA: ACM, 2002, pp. 1–3. [Online]. Available: http://doi.acm.org/10.1145/503272.503274

[12] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746

[13] G. Kiczales, "AspectJ(tm): Aspect-oriented programming in Java," in *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers*, ser. Lecture Notes in Computer Science, M. Aksit, M. Mezini, and R. Unland, Eds., vol. 2591. Springer, 2002. [Online]. Available: http://dx.doi.org/10.1007/3-540-36557-5_1

[14] S. Katz, "Transactions on aspect-oriented software development I," in *Transactions on Aspect-Oriented Software Development I*, A. Rashid and M. Aksit, Eds. Berlin, Heidelberg: Springer-Verlag, 2006, ch. Aspect Categories and Classes of Temporal Properties, pp. 106–134. [Online]. Available: http://dl.acm.org/citation.cfm?id=2168342.2168346

[15] D. Jin, P. O. Meredith, C. Lee, and G. Rosu, "Javamop: Efficient parametric runtime monitoring framework," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 1427–1430. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2012.6227231

[16] F. Chen and G. Rosu, "Parametric trace slicing and monitoring," in *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, 2009, pp. 246–261. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00768-2_23

[17] Y. Falcone, "You should better enforce than verify," in *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, ser. Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, Eds., vol. 6418. Springer, 2010, pp. 89–105. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16612-9_9

[18] M. Ducassé and E. Jahier, "Efficient automated trace analysis: Examples with morphine," *Electr. Notes Theor. Comput. Sci.*, vol. 55, no. 2, pp. 118–133, 2001. [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(04)00248-8

[19] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c - A software analysis perspective," in *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, ser. Lecture Notes in Computer Science, G. Eleftherakis, M. Hinchey, and M. Holcombe, Eds., vol. 7504. Springer, 2012, pp. 233–247. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33826-7_16

[20] Y. Falcone, K. Havelund, and G. Reger, "A tutorial on runtime verification," in *Engineering Dependable Software Systems*, ser. NATO Science for Peace and Security Series, D: Information and Communication Security, M. Broy, D. A. Peled, and G. Kalus, Eds. IOS Press, 2013, vol. 34, pp. 141–175. [Online]. Available: http://dx.doi.org/10.3233/978-1-61499-207-3-141

[21] K. Havelund and G. Reger, "Specification of parametric monitors," in *Formal Modeling and Verification of Cyber-Physical Systems, 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*, R. Drechsler and U. Kühne, Eds. Springer, 2015, pp. 151–189. [Online]. Available: http://dx.doi.org/10.1007/978-3-658-09994-7_6

[22] M. Chabot, K. Mazet, and L. Pierre, "Automatic and configurable instrumentation of C programs with temporal assertion checkers," in *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, 2015, pp. 208–217. [Online]. Available: http://dx.doi.org/10.1109/MEMCOD.2015.7340488

[23] S. Navabpour, Y. Joshi, C. W. W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister, "Rithm: a tool for enabling time-triggered runtime verification for C programs," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 603–606. [Online]. Available: http://doi.acm.org/10.1145/2491411.2494596

[24] R. Milewicz, R. Vanka, J. Tuck, D. Quinlan, and P. Pirkelbauer, "Lightweight runtime checking of C programs with RTC," *Computer Languages, Systems & Structures*, vol. 45, pp. 191–203, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2016.01.001